Django de A a Z

Crie aplicações web rápidas,
seguras e escaláveis com Python

Casa do Código | alura

TIAGO SILVA

Sumário

- ISBN
- Agradecimentos
- Sobre o autor
- Prefácio
- Sobre o livro
- Primeiros passos com Django
  - 1 Configuração do Python
  - 2 Primeiros passos com Django
- Admin e persistência de dados
  - 3 Trabalhando com Models
  - 4 Área administrativa
- Django avançado
  - 5 Trabalhando com Views e Urls
  - 6 Django ORM
  - 7 Trabalhando com templates - Parte I
  - 8 Trabalhando com templates - Parte II
  - 9 Trabalhando com forms - Parte I
  - 10 Trabalhando com forms - Parte II
  - 11 Trabalhando com autenticação
- Conteúdo extra
  - 12 Autenticação com redes sociais
  - 13 Serviços de e-mail
  - 14 Testes unitários
  - 15 Deploy no Heroku

ISBN

Impresso: 978-65-86110-68-5

Digital: 978-65-86110-67-8

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Dedico este livro primeiramente a Deus, pois foi Ele quem me concedeu chegar até aqui e realizou sonhos muito além daqueles que um dia pensei que poderia sonhar. Meu melhor Amigo e Pai, que me entende, me ouve e me ajuda, em todos os momentos de minha vida, sem Ele não sou e não desejo ser nada. A cada dia que se passa tenho me surpreendido mais com cada aprendizado que tenho adquirido através dEle e cada benção que tenho recebido.

À minha mãe Andréa que me criou e lutou para que eu me tornasse o homem que hoje sou, que passou por momentos difíceis para que eu pudesse sorrir e acreditou durante todo o tempo que eu chegaria até aqui, a verdadeira escritora dos meus livros. Posso até ser o autor que coloca conteúdo em cada página, mas foi ela que muitas vezes se sacrificou para que eu tivesse o que ela não pôde ter. Se hoje estou aqui, posso dizer que ela foi fundamental para isso e hoje nos orgulhamos um do outro por tudo que Deus nos permitiu conquistar juntos.

A meus pais França e Vilma, que me acompanharam e me acolheram em sua família desde pequeno, família esta que posso chamar de minha e que também foi fundamental para que eu alcançasse meus objetivos, sendo pessoas de referência em minha vida, me aconselhando o caminho certo por onde eu deveria ir, acreditando que eu seria capaz e se emocionando comigo.

À minha companheira, amiga, musa, namorada, noiva e esposa, meu eterno amor, minha inspiração e um dos motivos pelo qual busco ser uma pessoa melhor a cada dia, para permanecer sendo capaz de fazê-la se apaixonar por mim todos os dias, como me apaixono por ela em todos os instantes. Obrigado por estar ao meu lado cada segundo, acreditando em mim.

Aos meus irmãos, Diogo e Victor, que são exemplos que incansavelmente sempre segui na certeza de que me tornaria um grande e honrado homem, e à minha pequena e linda irmã Maria

Luiza, que com seu sorriso repleto e sincero sempre me traz luz e alegria, até mesmo em momentos complicados e conturbados que todos temos em nosso dia a dia.

À minha tia Janete e tio Eduardo, que investiram em meus estudos e assim me ajudaram a entrar para o ramo da tecnologia, e ao meu melhor amigo e irmão Marcos Abraão, que nesses mais de 15 anos de amizade sempre apostou alto em mim e investiu seu tempo e esforço.

Ao meu amigo Vinícius Albino, por ser essa pessoa que acrescenta em minha vida com seus conselhos, me ajudando a ser uma pessoa melhor e mais madura.

Ao meu padrasto Saulo, que junto de minha mãe Andréa auxiliou em minha criação, sempre me tratando como um filho, me aconselhando e ajudando em tudo que pôde. A todos os meus alunos, que são o real motivo e inspiração para que eu continue todos os dias buscando conhecimento para ser compartilhado entre todos de forma a ajudar no crescimento profissional e pessoal de cada um.

A toda a equipe da Casa do Código que pela segunda vez foi fundamental para que este livro viesse a ter o nível de qualidade que o leitor merece, em especial à Vivian Matsui que desde o primeiro livro acreditou em meu sonho e fez parte de todo o processo como minha editora-chefe. Que este seja o segundo de muitos livros que faremos juntos, se assim Deus permitir.

Ao meu amigo Fábio Porto, que me deu grande força e foi fundamental na melhoria da qualidade do livro através de sua leitura beta.

A todos vocês o meu muito obrigado, por fazerem parte da minha vida e acreditarem que eu me tornaria alguém cujo coração teria como missão compartilhar o aprendizado e sabedoria que graças a Deus vindo de vocês me tornou o homem que hoje sou.

Sobre o autor

Tiago Silva é graduado em Análise e Desenvolvimento de Sistemas, pós-graduado em Engenharia de Software, professor na área de tecnologia há mais de 8 anos e atualmente trabalha como engenheiro de software atuando diretamente com Desenvolvimento de Aplicações Web utilizando ferramentas do Python, como Django e Flask. Especialista em Python, JavaScript, Google Maps e Adobe Muse é fundador do Canal Digital Cursos e Canal do Prof, onde existem cursos inteiramente on-line focados em ajudar pessoas a entrarem para o mercado de trabalho.

Prefácio

Este livro foi criado com o intuito de partilhar com você, leitor ou leitora, mais sobre meus conhecimentos e experiências com o Django, um dos frameworks mais utilizados pelos programadores Python no mundo.

O conteúdo abordado aqui será bem direto ao assunto principal, com o objetivo de servir como um manual diário que terá tudo que é necessário para criar uma aplicação completa e de forma correta utilizando o Django como framework. Durante todo o livro, veremos imagens que expressam o resultado desejado da aplicação que construiremos, para ajudar você a acompanhar o caminho e chegar ao objetivo.

Fico alegre em saber que este livro é capaz de preparar pessoas para o mercado de trabalho onde o Python e o Django são muito utilizados. Com ele, você se tornará um completo especialista nessa ferramenta, sendo capaz de construir enormes aplicações que poderão ser utilizadas por pequenas, médias e grandes corporações.

Durante o andamento do livro, deixo algumas dicas para que você não precise passar por problemas pelos quais já passei no dia a dia com o uso do Django como ferramenta de trabalho, de modo que você tenha uma ótima produtividade enquanto estiver trabalhando com essa ferramenta.

O objetivo principal é que você consiga criar grandes aplicações em um curto tempo de forma rápida, consistente e segura. Faça bom proveito deste conteúdo. É um imenso prazer poder compartilhar com você o conhecimento que adquiri com muitas pessoas que desejavam ver meu crescimento profissional e assim faço também, compartilhando com outras pessoas que desejam crescer e alcançar seus sonhos. Desejo-lhe todo o sucesso do mundo.

Tiago Silva

Sobre o livro

Neste livro, você aprenderá um dos maiores frameworks de Python que existem na atualidade, o Django. Considerado entre os três frameworks web mais conhecidos e usados no Python, Django é uma ferramenta completa, robusta e de fácil implementação, que nos permite criar uma aplicação web de forma rápida e consistente.

A primeira versão oficial do Django foi publicada sob a licença BSD em 2005. Entre as principais características que o fazem ser um dos mais usados no mundo está o fato de ele ser seguro, fácil de aprender e de implementar em um ambiente Cloud (na nuvem).

Um breve resumo do mercado de trabalho do Django

A linguagem de programação Python está em alta no mercado de trabalho. Muitas empresas buscam profissionais com conhecimento em frameworks de Python, como Django, Flask, Tornado e muitos outros. A seguir, vemos uma lista com as principais empresas que usam o Django como framework para desenvolvimento de suas plataformas Web atualmente: Spotify, Instagram, YouTube, Dropbox, Bitbucket, Disqus, The Washington Post, Mozilla, Prezi, Pinterest, Reddit, Udemy, MIT, Coursera.

Como podemos ver, existem muitas empresas famosas que usam o Django como ferramenta em suas plataformas, sem contar as que o utilizam como ferramenta de trabalho. O objetivo principal aqui é preparar você para estar apto(a) a trabalhar com o Django e conseguir espaço dentro desse enorme mercado de Python que existe atualmente.

Para quem é este livro e quais os pré-requisitos

O público-alvo deste livro são pessoas que desejam aprender a criar aplicações web consideradas de grande porte, seguras, em um curto prazo e com facilidade.

Como pré-requisito é necessário que você tenha conhecimento em lógica de programação e na linguagem Python, não sendo necessário ter criado uma aplicação web com Python, mas conhecer sua lógica e sua sintaxe. É recomendável também que você consiga trabalhar com linhas de comando via terminal, pois usaremos alguns comandos do Django durante o decorrer do projeto em nosso livro.

O que aprenderei neste livro?

Você aprenderá a criar uma aplicação de grande porte, consistente e que interaja via client/server.

Criaremos uma aplicação web para consulta de médicos. Ela permitirá que usuários consultem os médicos mais próximos de sua localidade ou uma localidade específica, podendo filtrá-los por nome, especialidade, estado, cidade e bairro do médico.

Veremos no Django como criar um painel administrativo para a aplicação, além da criação de telas HTML também usando a tecnologia de templates do Django. Como complemento, veremos a criação de login via rede social e implementaremos um envio de notificações via serviço de e-mail.

Algo muito importante que veremos no livro será a ferramenta **Django ORM**, uma poderosa biblioteca que ele possui. Ela nos permite trabalhar de forma muito avançada com nosso banco de

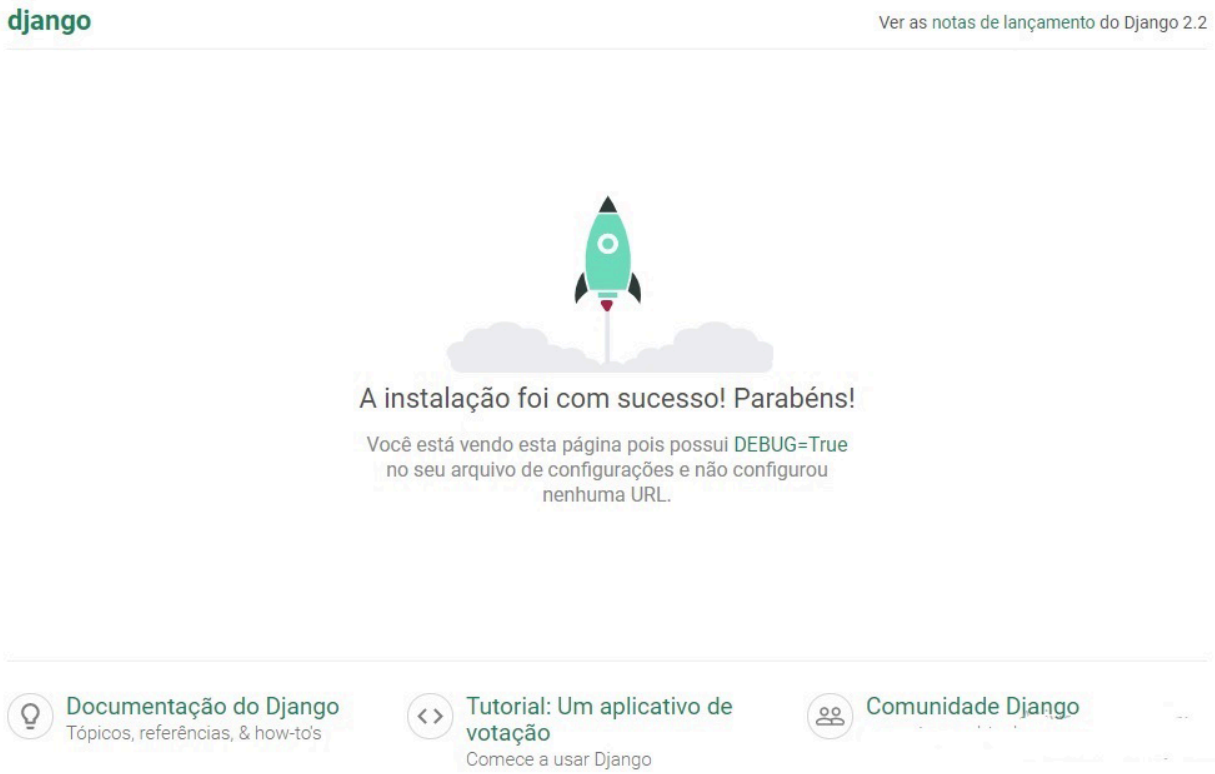
dados, através do conceito Mapeamento Objeto Relacional, utilizando o conceito de `models`.

Veremos também a manipulação de `views`, `templates` e `formulários`. Tudo de forma completa e concisa, com o principal objetivo de trazer a você um manual completo do Django, com tudo que você precisa saber para trabalhar com essa poderosa ferramenta.

Como estudar com este livro?

O livro foi escrito para ser estudado com a mão na massa, trazendo explicações bem sólidas sobre o assunto junto da execução prática de etapas de um sistema de busca de médicos, que será construído no decorrer dos capítulos.

Além das explicações contendo práticas bem elaboradas e de fácil entendimento, contamos com algumas observações e dicas em cada tema, conforme a minha experiência. São questões ou conselhos que deixarei para evitar que você passe por algum problema que já enfrentei utilizando o framework.



Primeiros passos com Django

Nesta unidade, veremos como fazer a instalação do Python e do Django, além de como configurar de forma completa e profissional uma aplicação feita em Django.

O objetivo é trazer um conhecimento sólido e consistente da forma correta de se configurar sua aplicação Django, diferente do que vemos em muitos casos, nos quais a aplicação não é configurada de forma profissional e segura.

Capítulo 1
Configuração do Python

Neste capítulo, veremos como fazer as instalações básicas do Python em nossa máquina e rodaremos nosso primeiro script para testar se tudo está dentro do esperado. Caso já tenha o Python em sua máquina, sinta-se à vontade para ir ao capítulo 2.

Vou fazer a demonstração utilizando os Sistemas Operacionais Windows e Linux (mais precisamente Ubuntu), mas tenha em mente que, se seu sistema operacional for qualquer outro que trabalhe baseado em Linux, ou se for macOS, os comandos de Linux provavelmente vão funcionar ou estarão bem próximos dele. A diferença é que o macOS utiliza o gerenciador de pacotes `brew` para fazer suas instalações.

1.1 Instalando o Python e suas dependências

Windows

Vá até o link https://www.python.org/downloads/ e faça o download da versão mais atual que estiver disponível. Clicando em `download`, isso provavelmente já ocorrerá direto. Uma dica é deixar o caminho de instalação exatamente onde o instalador deixa por padrão, pois colocá-lo em outro caminho pode impactar em questões de permissão de usuário etc.

> No momento da escrita do livro, a versão do Python era a 3.8.x (A versão 3.9.x até já está em uso, mas algumas bibliotecas ainda apresentam problema com ela, então recomendo que usem a 3.8.x).

Pronto, ao fazer isso seu Python já está instalado. Agora vamos configurar as variáveis de ambiente e instalar o gerenciador de pacotes do Python chamado `pip`.

Variáveis de ambiente do Python no Windows

Fique atento, pois o instalador do Python possui a opção de adicionar o `path` de instalação diretamente na variável de ambiente. Existem casos em que essa opção não existe, ou mesmo marcando-a pode ocorrer de o `path` não estar adicionado nas variáveis de ambiente. Caso isso ocorra siga os passos a seguir:

1. Abra o painel de controle e navegue até as configurações de sistema.
2. Selecione as configurações avançadas do sistema.
3. Clique em variáveis de ambiente.
4. Procure nas variáveis do sistema pela variável `path` .
5. Clique em `Editar` .
6. Veja se os valores `C:\Python38` e `C:\Python38\Scripts` já estão no campo de valor da variável de ambiente; se não existirem, adicione-os ao final da linha separados por ponto e vírgula (`;`). O `python37` , neste exemplo, é referente à pasta onde o Python foi instalado no seu sistema, então este valor pode ser diferente caso esteja instalando outra versão. Por exemplo, se for a versão `2.7.15` do Python, o valor será `Python27` ; se for `3.8.0` , o valor será `Python38` e assim por diante. Veja o caminho exato da sua instalação e substitua o valor acima.
7. Após isso, clique em `OK` .

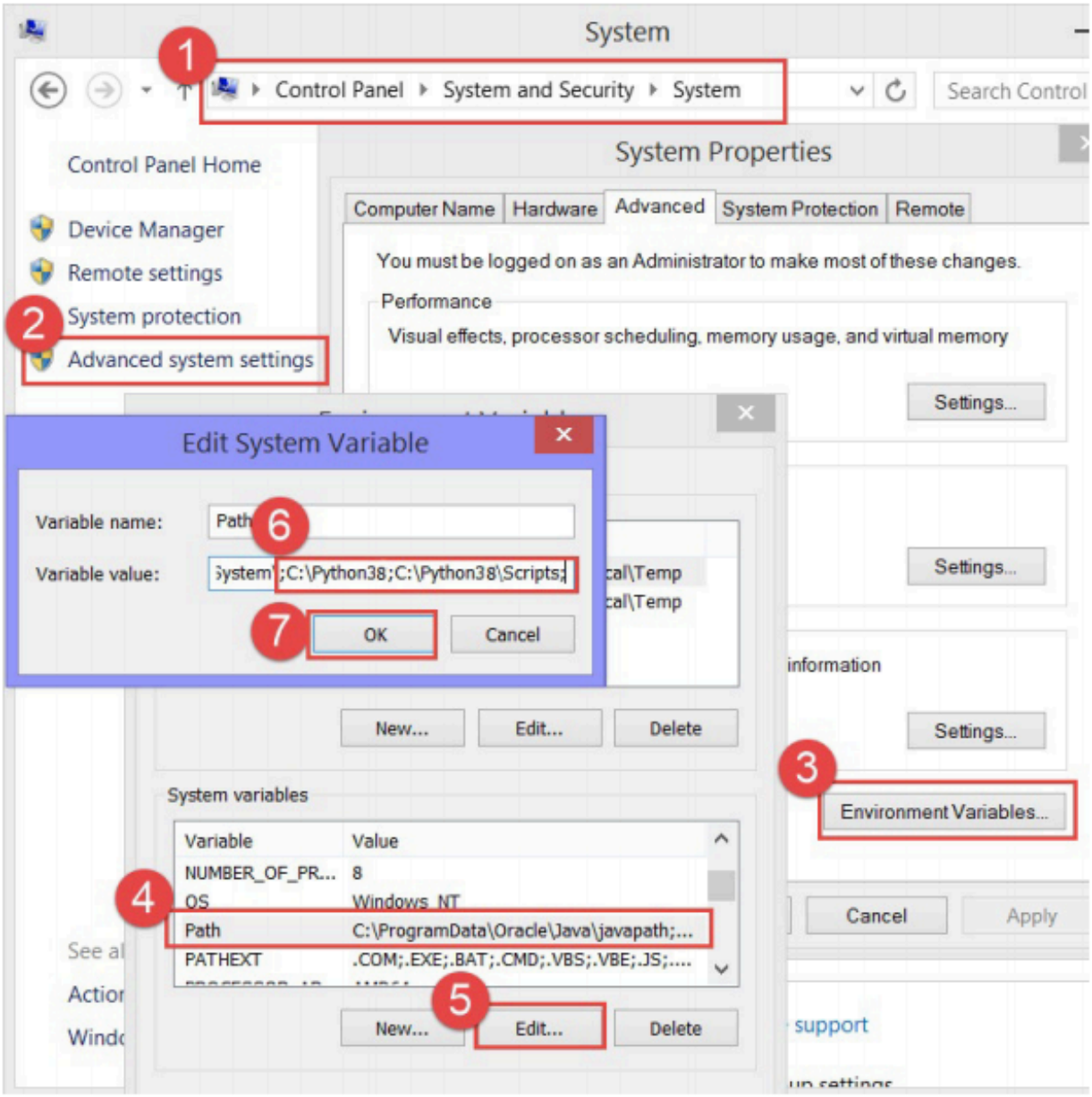


Figura 1.1: Painel de Controle do Windows

Instalando o PIP no Windows

O `pip` é um gerenciador de pacote que o Python possui. Com ele, conseguimos baixar novos pacotes/bibliotecas, gerenciá-los com recursos que permitem listar tudo que temos já instalado em nosso projeto e fazer a instalação desses pacotes com mais facilidade em outros ambientes.

A instalação do `pip` só funciona a partir da versão `2.7.9` do Python.

No menu do Windows, vá em `Executar`, abra seu terminal `cmd` e rode o comando a seguir para instalar o pacote do `pip`:

```
c:\Users\Tiago>python -m ensurepip
```

Ao término, ele retornará que todos os requisitos estão satisfeitos. Então, rode o comando a seguir:

```
c:\Users\Tiago>python -m ensurepip --upgrade:
```

Esse comando serve para realizar possíveis atualizações que não vieram no momento da instalação do `pip`.

Agora você possui o `pip` instalado e atualizado em seu computador.

Criando um virtualenv no Windows

O `virtualenv` é o ambiente virtual que o Python permite que você crie para poder trabalhar em diferentes versões do Python dentro de uma mesma máquina. Cada projeto pode lidar com uma versão diferente do Python graças a esse recurso. Dentro dele, fazemos as instalações de todas as bibliotecas que estamos utilizando dentro do projeto, assim podemos controlá-las de forma isolada dentro de um caso sem interferir em outro. Isso permite que tenhamos projetos mais seguros em questão de desenvolvimento.

Para instalar um `virtualenv` no Windows, rode o comando a seguir dentro do terminal:

```
c:\Users\Tiago>c:\Python37\Scripts\pip.exe install virtualenv
```

Não se esqueça de que, se sua versão for diferente, o caminho `python37` poderá ser diferente do mostrado aqui.

No local que preferir, crie uma pasta onde ficará o nosso primeiro projeto e chame-a de `hello_world`.

Dentro dessa pasta, rode o comando a seguir, para criar seu `virtualenv`. Assim poderemos realizar futuramente as instalações de pacotes necessários para o projeto sem afetar outros possíveis projetos que existirão em seu computador:

```
c:\Users\Tiago\Documents\hello_world>c:\Python37\Scripts\virtualenv.exe venv
```

Com seu `virtualenv` criado, é necessário que ele seja ativado. Pense nele da seguinte forma: o terminal que possuir o `virtualenv` ativado terá todas as bibliotecas daquele projeto ativadas nele. Para ativar seu `virtualenv`, rode o seguinte comando:

```
c:\Users\Tiago\Documents\hello_world>venv\Scripts\activate
```

Pronto, agora temos tudo preparado para iniciar nosso projeto. Seu terminal ficará parecido com o seguinte:

```
(venv) C:\Users\Documents\hello_world>
```

> Fique atento: se você fechar o terminal, precisará abrir outro e rodar esse comando novamente. Isso porque o ambiente, como dito, é virtual e precisa ser iniciado dentro do terminal que for utilizado toda vez que esse terminal for aberto pela primeira vez.

Ubuntu

Verifique primeiramente se você já possui o Python instalado por padrão em seu Ubuntu rodando o comando:

```
tiago_luiz@ubuntu:~$ which python
```

Provavelmente, ele retornará algo como `/usr/bin/python`, o que significa que o Python já está instalado. Mas se ele não retornar isso, e sim algo como `which: no python in`

`(/usr/local/sbin:/usr/local/bin:/usr/bin:/usr...)` , precisaremos instalá-lo, então siga os próximos passos.

Por meio do gerenciador de pacotes `apt-get` , rode o comando a seguir:

```
tiago_luiz@ubuntu:~$ sudo apt-get install python3.7
```

> No caso de ser outra versão, substitua o `python3.7` pela versão desejada.

O `pip` no Ubuntu é mais simples do que no Windows. Para rodá-lo, digite o comando a seguir, cuja função é a mesma que foi explicada no tópico sobre Windows:

```
tiago_luiz@ubuntu:~$ sudo apt-get install python-pip
```

O Python não exige variáveis de ambiente em sistemas Linux, então fique tranquilo, pois podemos pular essa parte.

Criando um virtualenv no Ubuntu

Como já explicamos no tópico de Windows o que é `virtualenv` , vamos direto para sua instalação e inicialização no projeto.

Para instalar um `virtualenv` no Ubuntu, rode o comando:

```
tiago_luiz@ubuntu:~$ sudo pip3 install virtualenv
```

Crie uma pasta onde preferir para ficar nosso primeiro projeto, e chame-a de `hello_world` .

Com a pasta criada, entre neste diretório pelo terminal e crie seu `virtualenv` através do comando a seguir:

```
tiago_luiz@ubuntu:~/hello_world$ virtualenv -p python3 venv
```

Agora, para que seu projeto rode com base neste `virtualenv` , você precisa executar o seguinte comando:

```
tiago_luiz@ubuntu:~/hello_world$ source ./venv/bin/activate
```

Agora, temos tudo pronto para iniciar nosso projeto. Seu terminal ficará da seguinte forma, ou algo próximo a isto, dependendo do caminho do projeto.

```
(venv) tiago_luiz@ubuntu:~/hello_world$
```

1.2 Escolhendo uma IDE

A escolha da IDE não é algo obrigatório, pois podemos utilizar até mesmo o bloco de notas para rodar Python, mas ter um ambiente para desenvolver a aplicação facilitará bastante nosso desenvolvimento. Algumas das IDEs a seguir são muito recomendadas e eu mesmo tenho vivência com todas as listadas:

- PyCharm (https://www.jetbrains.com/pycharm/)
- Visual Studio Code (https://code.visualstudio.com/)
- Atom (https://atom.io/)
- Sublime Text (https://www.sublimetext.com/)

> Todas são gratuitas. Fique atento apenas para o fato de que o PyCharm tem uma versão *community* gratuita e a versão *professional*, que é paga.

Neste livro, usaremos o Visual Studio Code.

1.3 Testando o ambiente para começar

Agora que temos tudo pronto, todo o processo está correto e sua máquina já está pronta para rodar o Python, vamos fazer nosso

primeiro script antes de iniciar a criação de nossa aplicação web com Django.

Com o `virtualenv` iniciado, dentro da pasta `hello_world`, crie um arquivo chamado `run.py`. Dentro do arquivo, escreva o código a seguir e, em seguida, vamos entender o que ele faz:

```
# -*- coding: utf-8 -*-
print('Olá Mundo')
```

A primeira linha de código é utilizada para dizer ao Python que ele deverá interpretar nossas Strings (textos) na codificação `utf-8`. Não entraremos nesse assunto de codificação neste livro, isso é apenas um teste para ver se o Python está instalado.

A segunda linha possui o método `print`, que é utilizado para exibir valores no terminal. Em nosso caso, o `print` exibirá o valor `Olá Mundo` apenas para testarmos se o Python está executando corretamente.

Agora, rode o comando a seguir para executar seu código:

```
(venv) tiago_luiz@ubuntu:~/hello_world$ python run.py
```

Você verá o seguinte resultado:

```
(venv) tiago_luiz@ubuntu:~/hello_world$ python run.py
Olá Mundo
(venv) tiago_luiz@ubuntu:~/hello_world$
```

Ao longo do livro, veremos mais a fundo o Django. Este primeiro capítulo foi apenas um manual para auxiliar na configuração do ambiente de desenvolvimento.

> Todos os códigos deste livro estão disponíveis no meu GitHub: https://github.com/tiagoluizrs/livro_django/.

Capítulo 2
Primeiros passos com Django

O Django é uma ferramenta robusta e completa muito utilizada no mercado de trabalho atual. Grandes empresas como Spotify, Instagram e YouTube utilizam esse framework como ferramenta de trabalho em seu dia a dia devido a diversos fatores que agilizam a produtividade e trazem segurança e flexibilidade durante a produção de funcionalidades de um sistema.

Neste capítulo, veremos os pilares fundamentais que toda aplicação Django precisa ter para que haja um bom e produtivo funcionamento do framework, tanto no momento do desenvolvimento quanto em teste e produção.

2.1 Instalando o Django

O Django é um framework do Python e para instalá-lo utilizamos o comando de instalação do `pip`.

Crie uma pasta chamada `livro_django`, onde criaremos nosso projeto Django, crie um `virtualenv` dentro da pasta igual ao que aprendemos no capítulo anterior, ative o `virtualenv` e entre na pasta pelo terminal. Após isso, rode o comando a seguir para instalar o Django em seu virtualenv.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install django
```

Perceba que a pasta do capítulo anterior se chamava `hello_world`, pois estávamos somente fazendo testes. A versão que estamos usando é a 3.1.5. Este livro funciona bem para as versões 2 e 3. Mesmo que sua versão 3 seja mais nova, provavelmente não haverá problemas para usar.

Com tudo instalado, já podemos começar a criar nosso primeiro projeto no Django e, para isso, precisamos seguir alguns passos.

2.2 Criando um projeto no Django

Ao instalar o Django em nosso virtualenv, nós habilitamos alguns scripts providos dele que nos permitem fazer coisas, como criar um app dentro do projeto (você verá mais à frente o que são apps), realizar uma migração/atualização da estrutura do banco de dados e daí em diante.

Nesse momento, vamos usar o comando `django-admin` para criar nossos arquivos iniciais do projeto. Rode o comando a seguir para criar os arquivos principais. Caso seu sistema operacional seja Windows, dê uma olhada na dica que está logo adiante.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ django-admin startproject
medicSearchAdmin . (<-- Obs: Este ponto precisa ser colocado)
```

Um ponto a ser observado é que dependendo da sua configuração do Windows, o comando `django-admin` precisa ter a extensão `.exe`, ficando assim: **django-admin.exe**.

Outro ponto a se analisar é que após `medicSearchAdmin` existe um ponto final. Ele precisa estar no comando para que o app `medicSearchAdmin` seja criado no mesmo diretório em que estamos, que é a raiz do projeto.

Após rodarmos o comando, teremos uma estrutura parecida com a seguinte:

```
livro_django
├──medicSearchAdmin
│       __init__.py
│       asgi.py
│       settings.py
│       urls.py
│       wsgi.py
├──venv
├──manage.py
└──requirements.txt
```

Fique tranquilo, ao longo do capítulo essa estrutura fará mais sentido para você. Se ainda assim você tiver alguma dúvida e quiser ampliar a visão sobre a estrutura do projeto, acesse o link dele no GitHub para compará-lo com o da estrutura vista agora. https://github.com/tiagoluizrs/livro_django.

2.3 Regra de negócios do sistema

Para termos uma visão mais ampla do contexto do projeto que vamos desenvolver, é importante conhecermos as regras de negócio

dele. Elas serão fundamentais para que tenhamos um padrão correto durante o desenvolvimento do projeto e possamos realizar uma entrega de qualidade do nosso sistema. A seguir, deixamos listadas todas as regras de negócio do sistema de busca de médicos que criaremos.

Usuários e permissões

As funções administrativas têm como finalidade gerenciar o que cada usuário pode fazer dentro do sistema. Permissões de edição e remoção de usuários que são concedidas ao administrador em hipótese alguma devem ser atribuídas ao médico ou paciente, por exemplo. Em nosso sistema, teremos as três seguintes funções de usuários:

- Admin: possui permissão em todas as áreas do sistema.
- Médico: pode editar seu perfil, adicionando especialidades e locais de trabalho a ele.
- Paciente: pode editar seu perfil, adicionando preferências de localidades e especialidades e poderá realizar buscas de médicos no sistema.

Telas

- Admin:
    - Gerenciamento de usuários.
    - Gerenciamento de especialidades.
    - Gerenciamento de tipos de usuários.
    - Gerenciamento de permissões de telas.
    - Vê todas as telas que o médico e o paciente veem.
- Médico:
    - Gerenciamento do próprio perfil.
    - Vê todas as telas que o paciente vê.
- Paciente:
    - Login.
    - Gerenciamento do próprio perfil.
    - Busca de Médicos.

- Favoritos.

2.4 Arquivos de configuração do projeto

O arquivo de configuração é um dos elementos de maior importância em nosso projeto, pois é através dele que faremos todas as configurações. Em nossa aplicação, teremos quatro arquivos de configuração, sendo um deles o arquivo base, que conterá tudo que é necessário em qualquer ambiente de projeto, seja ele produção, teste, desenvolvimento etc. Os outros três são para dev, teste e produção. Mesmo que não usemos todos, será importante aprender a criar todos eles, como se estivéssemos em um projeto real.

Para isso, pegue a estrutura do projeto atual e, dentro da pasta `medicSearchAdmin`, crie outra, chamada `settings`. Dentro dela, adicione os arquivos `development.py`, `production.py`, `testing.py` e `__init__.py`. Agora que criamos esses arquivos, pegue o arquivo `settings.py`, que está na pasta `medicSearchAdmin`, e coloque-o também na pasta `settings`. Fazemos isso para que dentro do arquivo `settings.py` fiquem apenas elementos que serão utilizados em qualquer ambiente, seja produção, teste ou desenvolvimento. Porém, tudo o que precisarmos criar de modo diferente para cada ambiente, colocaremos nos arquivos que acabamos de criar.

Um bom exemplo é o banco de dados. Geralmente, usamos o `sqlite` localmente (inclusive faremos isso no livro), mas em teste é comum termos um banco de dados relacional mais robusto, como MySQL ou Postgre, enquanto em produção costumamos ter outro banco de dados, assim os dados de teste não se misturam com os dados de produção. Para fazermos esse tipo de separação, colocaríamos as configurações do banco de dados em seus respectivos arquivos (`development.py`, `testing.py` e `production.py`). Durante o decorrer do livro, veremos mais sobre o uso desses três arquivos. Veja a seguir como deverá ficar a estrutura do seu projeto:

```
livro_django
├──medicSearchAdmin
│        settings
│        │       __init__.py
│        │       settings.py
│        │       development.py
│        │       production.py
│        │       testing.py
│        │
│        __init__.py
│        urls.py
│        wsgi.py
├──venv
├──manage.py
└──requirements.txt
```

Agora precisaremos abrir cada arquivo e adicionar algumas configurações a eles, vamos lá.

medicSearchAdmin/settings/development.py

```
from .settings import *

DEBUG = True

# Crie a secret key para seu ambiente de desenvolvimento
SECRET_KEY = 'ixb62ha#ts=ab4t2u%p1_62-!5w2j==j6d^3-j$!z(@*m+-h'

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Se você preferir, toda vez que o import começar com . no caminho, você pode usar o nome do projeto. Um exemplo básico: `from .settings import *` --> `from livro_django.settings import *`. Nele, `livro_django` é o nome do projeto, ou seja, a pasta raiz.

medicSearchAdmin/settings/testing.py

```
from .settings import *

DEBUG = True

# Crie a secret key para seu ambiente de teste
SECRET_KEY = 'ixb6fh&#ts=&bt$au%pgp_62-!8dw2j==j)d^3-j$!z(@*m+-h'

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

medicSearchAdmin/settings/production.py

```
from .settings import *

DEBUG = False

# Crie a secret key para seu ambiente de produção
SECRET_KEY = 'ixb6fha#ts=&b4t2u%p1_62-!8dw2j==j)d^3-j$!z(@*m+-h'

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Por ora, não alteraremos o banco de dados da aplicação, mas fique à vontade para criar uma `SECRET_KEY` diferente para cada arquivo. Não se esqueça de abrir o arquivo `medicSearchAdmin/settings/settings.py` e remover dele a configuração do banco de dados. É só procurar por `DATABASES` e remover a variável e as configurações feitas nela dentro desse arquivo. Vale lembrar que estamos fazendo isso porque adicionamos uma configuração de `database` para cada arquivo que criamos e, se deixarmos a configuração também no arquivo `settings.py`, poderemos confundir os colegas de equipe em um projeto real.

> Algo que eu costumo fazer quando uso repositórios públicos do git é adicionar os arquivos de configuração do projeto ao arquivo `.gitignore`, para que eles não subam junto aos demais para o repositório, tendo em vista que dados como `SECRET_KEY` não devem ser compartilhados em repositórios públicos.

Eu vou deixar a seguir o código de configuração para se conectar em cada um dos seguintes bancos de dados: PostgreSQL, MariaDB, MySQL, SQLServer e Oracle.

PostgreSQL

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install psycopg2
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'Nome do seu banco de dados',
        'USER': 'Nome do usuário',
        'PASSWORD': 'Senha',
        'HOST': 'Nome do host de conexão ao banco',
        'PORT': 'Número da porta de comunicação',
    }
}
```

MariaDB ou MySQL

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install mysqlclient
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'Nome do seu banco de dados',
        'USER': 'Nome do usuário',
        'PASSWORD': 'Senha',
        'HOST': 'Nome do host de conexão ao banco',
        'PORT': 'Número da porta de comunicação',
    }
}
```

SQLServer

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install django-pyodbc-azure
```

```
DATABASES = {
    'default': {
        'ENGINE': 'sql_server.pyodbc',
        'NAME': 'Nome do seu banco de dados',
        'USER': 'Nome do usuário',
        'PASSWORD': 'Senha',
        'HOST': 'Nome do host de conexão ao banco',
        'PORT': 'Número da porta de comunicação',
    },
}
```

Oracle

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'Nome do seu banco de dados',
        'USER': 'Nome do usuário',
        'PASSWORD': 'Senha',
        'HOST': '',
        'PORT': '',
    }
}
```

```
# Se você não usar um arquivo tnsnames.ora e for realizar a conexão usando
o SID, preencha o HOST e o PORT:
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'Nome do seu banco de dados',
        'USER': 'Nome do usuário',
        'PASSWORD': 'Senha',
        'HOST': 'Nome do host de conexão ao banco',
        'PORT': 'Número da porta de comunicação',
    }
}
```

Não esqueça que, além da instalação que fazemos da biblioteca e a configuração no projeto, precisamos ter também instalado em nosso computador o banco que estamos querendo configurar, ou pelo menos o driver de conexão do banco, para que o Python consiga instalar a biblioteca compatível com ele e assim realizar a conexão.

Esse trecho de cima não vai fundo em como realizar a conexão em cada banco de dados, mas para mais informações você pode consultar a documentação oficial. Lá você pode identificar quais passos são necessários antes de configurar o banco no projeto de forma específica para cada tipo de database: https://docs.djangoproject.com/en/3.1/ref/databases/.

Dois elementos muito importantes e que devem ser alterados no arquivo principal de `settings` são o `TIME_ZONE` e o `LANGUAGE_CODE`. Abra o arquivo `settings.py` que está na pasta `medicSearchAdmin/settings` e altere-o conforme o exemplo a seguir:

medicSearchAdmin/settings/settings.py

```
LANGUAGE_CODE = 'pt-br'

TIME_ZONE = 'America/Sao_Paulo'
```

Nesse mesmo arquivo, precisamos configurar o local em que ficarão nossos arquivos estáticos. Para isso, vá até o final do arquivo e abaixo da linha `STATIC_URL = '/static/'` adicione o código a seguir:

medicSearchAdmin/settings/settings.py

```
from pathlib import Path
# Adicione a linha a seguir no topo do arquivo após a importação da lib pathlib
import os

STATIC_URL = '/static/'
# Adicione apenas a linha abaixo
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Agora precisamos configurar os IPs que estarão liberados para rodarem nossa aplicação.

Neste mesmo arquivo temos a variável `ALLOWED_HOSTS`, que é uma lista do Python. Quando `DEBUG` estiver `True` e `ALLOWED_HOSTS` estiver vazio, ele liberará nosso endereço local. Um exemplo básico de configuração dessa variável seria `['localhost', '127.0.0.1', '[::1]']`.

> Se estivermos em produção, `ALLOWED_HOSTS` precisará ser o endereço de IP da máquina que executará nossa aplicação. Se passarmos o valor ["*"] para a variável, liberaremos qualquer endereço de `IP` ou `domínio` para rodar nossa aplicação. Essa configuração não é recomendável para o ambiente em produção.

Como precisamos liberar IPs para ambientes diferentes, nesse caso vamos retirar a variável `ALLOWED_HOSTS` do arquivo `medicSearchAdmin/settings.py` e adicioná-la separadamente nos arquivos `development.py`, `production.py` e `testing.py`. Adicione essa variável abaixo da `SECRET_KEY` para ficar organizado. Vamos lá.

medicSearchAdmin/settings/development.py

```
SECRET_KEY = 'ixb62ha#ts=ab4t2u%p1_62-!5w2j==j6d^3-j$!z(@*m+-h'

# Você também pode deixar em branco com colchetes vazios []
ALLOWED_HOSTS = ['127.0.0.1']
```

medicSearchAdmin/settings/testing.py

```
SECRET_KEY = 'ixb6fh&#ts=&bt$au%pgp_62-!8dw2j==j)d^3-j$!z(@*m+-h'

# Alterar para o ip do ambiente de teste quando houver.
ALLOWED_HOSTS = ['127.0.0.1']
```

medicSearchAdmin/settings/production.py

```
SECRET_KEY = 'ixb6fha#ts=&b4t2u%p1_62-!8dw2j==j)d^3-j$!z(@*m+-h'

# Alterar para o ip do ambiente de produção quando houver.
ALLOWED_HOSTS = ['127.0.0.1']
```

Por fim, nossa aplicação precisará olhar para esses arquivos de configuração. Para isso, altere o arquivo `manage.py` que fica na raiz do projeto. Abra o arquivo e altere-o igual ao exemplo a seguir:

manage.py

```
# Troque a linha
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'settings')

# Por esta linha
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
'medicSearchAdmin.settings.development')
```

Com isso, estamos dizendo ao Django que ele deverá olhar para o arquivo `medicSearchAdmin.settings.development.py` sempre que não houver uma variável de ambiente `DJANGO_SETTINGS_MODULE` configurada. Mais adiante, veremos como podemos alterá-la para olhar os arquivos de `production` e `testing`.

Agora que temos tudo certo, partiremos para a etapa na qual rodaremos o primeiro `runserver`. Vamos lá.

2.5 Nosso primeiro Run

Com o projeto configurado corretamente, vamos usar o comando a seguir para rodar nossa aplicação.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py runserver
127.0.0.1:8080
```

Teremos uma saída similar a esta:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py runserver
127.0.0.1:8080
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
November 26, 2019 - 12:06:15
Django version 2.2.7, using settings 'medicSearchAdmin.settings'
Starting development server at http://127.0.0.1:8080/
Quit the server with CTRL-BREAK.
```

Se acessarmos nossa aplicação, veremos algo similar a isto:

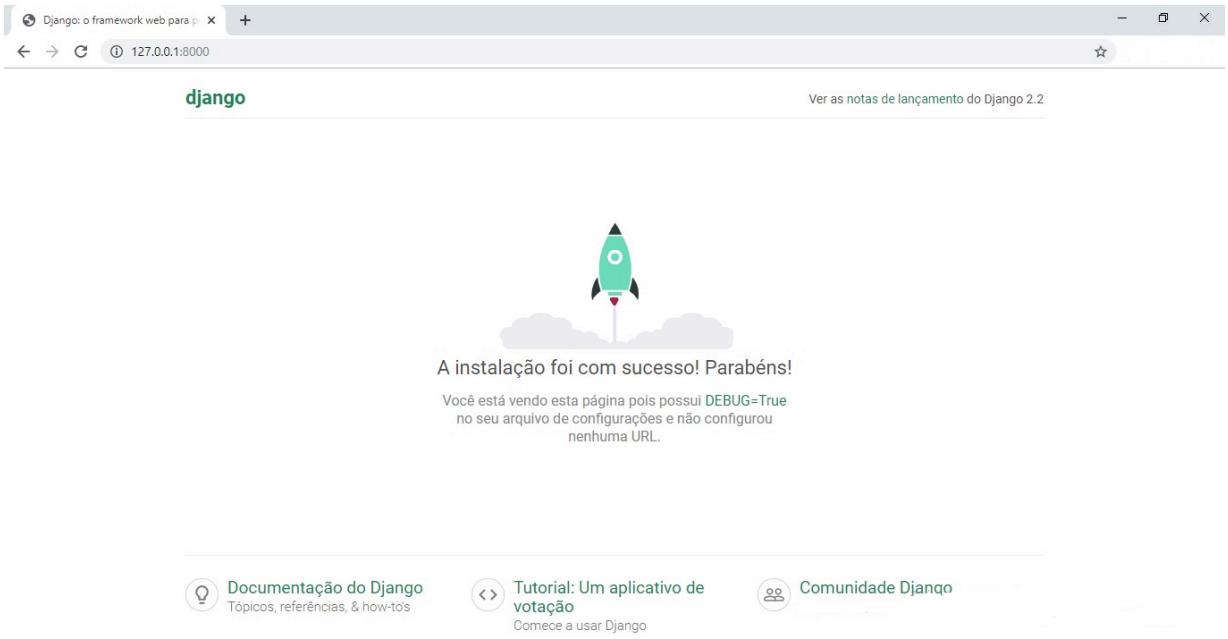


Figura 2.1: Primeiro Run

Como vimos anteriormente, deixamos configurado o arquivo `medicSearchAdmin.settings.development.py` como padrão. Se quisermos alterar isso, precisamos setar o arquivo que desejamos como principal da configuração antes de rodarmos o comando `runserver`. Vamos ver no exemplo a seguir:

Windows

```
(venv) C:\Users\tiago_luiz\livro_django> set
DJANGO_SETTINGS_MODULE=medicSearchAdmin.settings.production
(venv) C:\Users\tiago_luiz\livro_django> python manage.py runserver
127.0.0.1:8080
```

Ubuntu/Mac

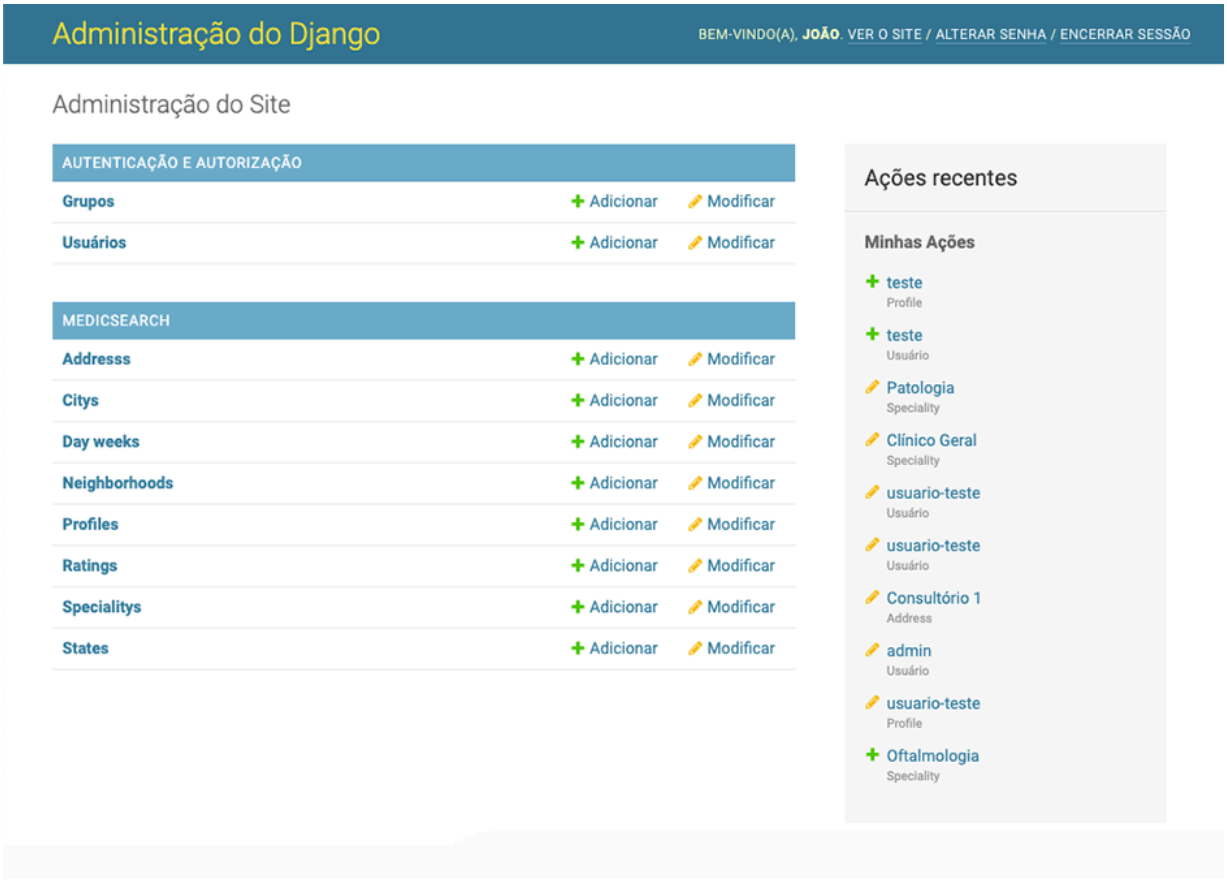
```
(venv) tiago_luiz@ubuntu:~/livro_django$ export
DJANGO_SETTINGS_MODULE=medicSearchAdmin.settings.production
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py runserver
127.0.0.1:8080
```

Após alterarmos a variável de ambiente responsável por armazenar o arquivo de configuração do nosso projeto, poderemos rodar qualquer um dos outros ambientes, seja produção, teste ou desenvolvimento, apenas alterando nesse lugar.

Vale lembrar que você também pode criar outros arquivos de configuração com outros nomes, seguindo os padrões que você preferir, sendo possível ter mais configurações de ambiente em cada arquivo.

Neste capítulo, foi possível compreender a forma correta e profissional de criação do ambiente de um projeto Django, onde podemos criar arquivos de configuração distintos para cada finalidade que temos, seja ela produção, teste ou desenvolvimento.

Também conseguimos compreender as regras de negócio do projeto e a forma correta de rodá-lo em nossa máquina. Em nossa próxima unidade, veremos como configurar nossas models e nosso painel administrativo.



Admin e persistência de dados

Esta é a unidade em que aprenderemos a configurar nosso painel administrativo e a personalizar a estrutura da tabela de usuários do Django, que embora já venha padronizada, é possível criarmos elementos dentro dela que estejam mais voltados para nossa regra de negócio. Veremos também permissões de usuário e a construção das models de nossa aplicação.

Capítulo 3
Trabalhando com Models

Neste capítulo, aprenderemos a configurar nossas `models`, que mais para a frente se tornarão telas do painel administrativo onde

poderemos gerenciar os dados que salvaremos em nosso banco de dados.

Como sabemos, `models` são classes responsáveis por representar nossas tabelas do banco de dados dentro de nossa aplicação. São as `models` que nos permitem fazer manipulação de dados em nosso banco. É através da model que realizamos leitura e escrita de dados em nossa base. Com o Django é possível manipular toda a base de dados através da configuração correta das nossas classes `Models`. A seguir, veremos como configurar nossa estrutura de banco de dados para trabalhar com nossas `models`.

3.1 Configurando nossa estrutura de banco de dados

Antes de começarmos a configurar nossas `models`, precisamos rodar nossa primeira migração e criar nosso superadmin. Vamos lá.

Vamos aos passos para realizar a criação das tabelas da nossa aplicação.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
```

Após fazer isso, você verá um resultado similar ao do exemplo a seguir em seu terminal.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
```

```
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying sessions.0001_initial... OK
```

Isso significa que você conseguiu criar as tabelas padrões de uma aplicação Django. Parabéns, o primeiro passo já foi dado! Vamos adiante.

Agora que temos a estrutura principal de banco de dados, vamos criar um usuário superadmin que poderá realizar qualquer tarefa em nossa aplicação. Para isso, rode o seguinte comando:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py createsuperuser
```

Esse comando pedirá um nome de usuário, um e-mail e uma senha. Preencha tudo conforme solicitado e você verá um resultado similar ao do exemplo a seguir em seu terminal.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py createsuperuser
Usuário (leave blank to use 'tiago.silva'): admin
Endereço de email: email@email.com
Password:
Password (again):
Esta senha é muito curta. Ela precisa conter pelo menos 8 caracteres.
Esta senha é muito comum.
Esta senha é inteiramente numérica.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Podemos ver que em um dado momento ele diz que a senha é fraca e me pergunta se quero criar mesmo assim. Eu aceitei, pois estamos em um ambiente de desenvolvimento, porém, em um ambiente de produção, sempre crie uma senha fortemente segura.

Pronto. Podemos rodar nossa aplicação e acessar nosso painel administrativo. Agora criaremos nosso primeiro app no Django.

3.2 Criando um app

O Django possui o conceito de app, segundo o qual podemos criar vários apps dentro de nossa aplicação. Assim é possível que cada app dentro do sistema tenha sua finalidade específica. Em um sistema muito grande, por exemplo, podemos ter um app responsável pela contabilidade, enquanto outro é responsável pela parte de RH e os dois podem se cruzar no meio do caminho graças à Orientação a Objetos. Como nosso sistema é mais simples, faremos um único app que conterá tudo o que precisamos.

Para criarmos nossa primeira aplicação, executaremos o seguinte comando em nosso terminal:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py startapp
medicSearch
```

Se pararmos para olhar nosso projeto, veremos que uma nova pasta com outra estrutura foi criada. Veja só:

```
livro_django
├──medicSearch
│        migrations
│        __init__.py
│        admin.py
```

```
│        apps.py
│        models.py
│        tests.py
│        views.py
├──medicSearchAdmin
│        settings
│        │       settings.py
│        │       development.py
│        │       production.py
│        │       testing.py
│        │
│        __init__.py
│        urls.py
│        wsgi.py
├──venv
├──manage.py
└──requirements.txt
```

No decorrer do livro, essa estrutura será um pouco alterada para que tenhamos uma aplicação mais escalável, mas fique calmo, faremos isso gradativamente.

Agora, precisamos adicionar esse app para ser gerenciado por nosso admin. Abra o arquivo `settings.py` dentro da pasta `medicSearchAdmin/settings` e altere o código como no exemplo a seguir:

```
# Adicione 'medicSearch' como um item na lista dos INSTALLED_APPS
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'medicSearch'
]
```

Dentro do diretório `medicSearch`, crie um novo diretório chamado `models` e, dentro dele, um arquivo chamado `__init__.py`. Após isso, pegue o arquivo `models.py` que está dentro do diretório `medicSearch` e apague-o. O que muitos fazem é criar todas as models dentro de um único arquivo, mas isso não é recomendável, nós criaremos um arquivo de model para cada tabela que desejamos criar. Se ainda não sabe o que é model, fique tranquilo, abordaremos esse assunto no próximo capítulo mais profundamente, mas, basicamente, cada model será também uma tabela do nosso banco de dados.

Após fazer essa alteração, a estrutura deverá ficar assim:

```
livro_django
├──medicSearch
│        migrations
│        models
│        │     __init__.py
│        │
│        __init__.py
│        admin.py
│        apps.py
│        tests.py
│        views.py
├──medicSearchAdmin
│        settings
│        │       settings.py
│        │       development.py
│        │       production.py
│        │       testing.py
│        │
│        __init__.py
│        __db.sqlite3
│        urls.py
│        wsgi.py
├──venv
├──manage.py
└──requirements.txt
```

Nossa primeira model será a de perfil. Vamos criá-la agora. Dentro do diretório `medicSearch/models`, crie o arquivo `Profile.py` e vamos colocar a mão na massa!

Abra o arquivo `Profile.py` e adicione o código a seguir:

medicSearch/models/Profile.py

```
from medicSearch.models import *

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    role = models.IntegerField(choices=ROLE_CHOICE, default=3)
    birthday = models.DateField(default=None, null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    token = models.CharField(max_length=255, null=True, blank=True)

    def __str__(self):
        return '{}'.format(self.user.username)

    @receiver(post_save, sender=User)
    def create_user_profile(sender, instance, created, **kwargs):
        try:
            if created:
                Profile.objects.create(user=instance)
        except:
            pass

    @receiver(post_save, sender=User)
    def save_user_profile(sender, instance, **kwargs):
        try:
            instance.profile.save()
        except:
            pass
```

Vamos entender o código:

- **from medicSearch.models import**: aqui nós importamos todas as models que existirem dentro do diretório `models`. Por

enquanto, temos apenas o arquivo `Profile.py`, futuramente teremos diversos outros.

- `__str__` : método que usamos para exibir o objeto quando quisermos acessá-lo através de sua instância.
- **create_user_profile**: usaremos esse método para criar o perfil do usuário. Para que isso ocorra automaticamente, precisamos passar a notação `@receiber(post_save, sender=User)`, onde `post_save` informa que, em um determinado método `post`, `create_user_profile` deve ser acionado. Já `sender=User` informa qual classe `Model` está sendo chamada no `post`, sendo assim, quando houver um `post` na classe `User`, deverá ser chamado o método `create_user_profile`. Não podemos esquecer que requisições `post` são para criações, então um `post` na classe `User` aciona o método `save` dela. No resumo, toda vez que ocorrer a criação de um novo usuário, um perfil será criado para ele.
- **save_user_profile**: salva qualquer alteração no perfil automaticamente quando entramos no painel de perfil e alteramos qualquer dado.

Fique atento! O primeiro usuário do sistema é criado pelo comando `createsuperuser`, ou seja, ele não terá um perfil de usuário, por isso colocamos um `try` e `except` em nossos métodos `create_user_profile` e `save_user_profile`, para que a aplicação não quebre no momento da realização de login com o admin que ainda não possui perfil.

3.3 Tipos de dados e campos

O Django ORM é um assunto que abordaremos mais à frente, mas por ora precisamos conhecer os tipos de dados principais que uma model do Django consegue comportar. Para esses tipos de dados, o

Django reserva um objeto diferente, que será representado como um campo de formulário dentro do painel admin. Entre eles, estão:

- **CharField**: campos de textos de até 255 caracteres. No admin, aparecerá como um `input text`.
- **TextField**: campos de textos de tipo small, medium e long text. No admin, aparecerá como um `textarea`.
- **IntegerField**: campo de inteiros exibido como `input number` no Django.
- **DecimalField**: campo de números decimais, flutuantes, exibido como `input number` no Django.
- **DateField**: campo de data representado pelo campo de `date_field` no `html`.
- **BooleanField**: campo de booleano representado como um `input checkbox` no Django.
- **ImageField**: campo de upload de imagem representado como um `input file`, porém só aceita imagem.
- **FileField**: campo de upload de imagem representado como um `input file`. Aceita qualquer formato de arquivo que o html suporte.
- **ManyToManyField**: campo que representa a relação de muitos para muitos do banco de dados. É representado por um `select` de múltiplos elementos no admin do Django. O método `ManyToManyField` recebe como parâmetro obrigatório o objeto que será relacionado com a model em que o `ManyToManyField` está sendo criado.
- **OneToOneField**: campo que representa a relação de um para um do banco de dados. É representado por um `select` de um elemento no admin do Django. O método `OneToOneField` recebe como parâmetro obrigatório o objeto que será relacionado com a model em que o `OneToOneField` está sendo criado.
- **ForeignKey**: campo de relacionamento padrão com apenas a chave de `foreing key` da tabela, similar ao OneToOne. Também representado por um campo de `select` de um elemento. O método `ForeignKey` recebe como parâmetro obrigatório o objeto

que será relacionado com a model em que o `ForeignKey` está sendo criado.

> A diferença entre `OneToOne` e `ForeingKey` na model é que `OneToOneField` (one-to-one) realiza, na Orientação a Objetos, a noção de composição, enquanto ForeignKey (one-to-many) se refere à agregação (Ambos são conceitos da Orientação a Objetos).

Existem também alguns parâmetros que podemos passar dentro desses fields que são muito úteis, entre eles:

- **default**: seta o valor padrão quando nada for colocado.
- **choice**: este parâmetro serve para facilitar nosso trabalho. Quando temos um campo de inteiro, decimal ou texto, podemos criar um `choice`, onde adicionamos como parâmetro uma tupla. Ela substituirá o campo de texto ou número por um select com as opções permitidas para o campo. Vamos ver um exemplo com a `ROLE_CHOICE`:
  - `ROLE_CHOICE = ((1, 'Admin'),(2, 'Médico'),(3, 'Paciente'))`. Vamos adicionar essa tupla no arquivo principal das nossas models, fique tranquilo.
- **null**: parâmetro que permite que o campo seja nulo no banco de dados.
- **blank**: parâmetro que diz que nosso campo do formulário, que representa a coluna no banco de dados, poderá ficar em branco ao ser salva. Mas tome cuidado, pois se o parâmetro `null` estiver como `False`, um erro poderá ser disparado, pois não podemos dizer que um campo que é `NOT NULL` pode ficar em branco no formulário.
- **on_delete**: usado nos campos `ManyToManyField`, `OneToOneField` e `ForeignKey`. Geralmente usado para atribuir ao relacionamento a ação de CASCADE do banco de dados.
- **max_length**: tamanho máximo que um campo pode ter.

- **related_name**: usado quando criamos um relacionamento (`ManyToManyField` , `OneToOneField` ou `ForeignKey`), para o Django identificar o campo.

Agora precisamos adicionar nossa model ao arquivo `__init__.py` . Ele será o responsável por comunicar ao Django quais models deverão refletir a estrutura de nossa base de dados. Isso é usado para quando não quisermos que algo seja migrado em determinado momento, dessa forma, simplesmente não colocamos a model no `__init__.py` e ela não será migrada naquele momento. Abra o arquivo `__init__.py` dentro do diretório `medicSearch/models` e adicione o código a seguir:

medicSearch/models/__init__.py

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

ROLE_CHOICE = (
    (1, 'Admin'),
    (2, 'Médico'),
    (3, 'Paciente')
)

from .Profile import Profile
```

As quatro primeiras linhas são bibliotecas que estamos importando para que todas as models que forem importadas depois delas possam usá-las sem precisar importar separadamente em seus arquivos. A aplicação Django olhará para o `__init__.py` que está na pasta `models` e, desse modo, serão identificadas quais as models habilitadas no sistema. Se uma model não estiver nesse arquivo, ela não estará em uso no sistema. Isso pode acontecer às vezes se você ainda não tiver terminado de criar a model, assim ela não deverá ser colocada no `__init__.py` .

Logo depois vemos a `ROLE_CHOICE` que estamos chamando dentro do arquivo `Profile.py`. Nós até falamos sobre essa variável quando vimos sobre os parâmetros que podem ser colocados em cada tipo de campo da model.

Por fim, temos o `from .Profile import Profile`, que é onde importamos a model `Profile` dizendo ao Django que ele será uma tabela de nosso banco de dados. Toda vez que criarmos uma nova model e quisermos que ela se torne uma tabela de nosso banco de dados, precisamos importá-la no arquivo `__init__.py`, um abaixo do outro.

Rodaremos novamente a migração para que seja criada a tabela de Profile em nossa aplicação. Como essa tabela não existia no banco e nem a `model` em nosso projeto, precisamos rodar um comando anterior ao migrate, o `makemigrations`. Esse comando do Django cria um novo arquivo com todas as alterações feitas nas models ativas do sistema (models ativas são as setadas no arquivo `__init__.py`). Se uma model foi criada, alterada ou removida, ele escreverá em um novo arquivo de migração. Repare que dentro da pasta `migrations`, que fica no diretório `medicSearch`, vão aparecendo novos arquivos de migração com as mudanças solicitadas. Após rodar o comando `makemigrations`, rodaremos sempre o comando `migrate` e ele identificará um novo arquivo dentro da pasta `migrations` e rodará as modificações escritas dentro desse arquivo realizando as alterações efetivas no banco de dados. Vamos lá.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py makemigrations
medicSearch
```

Ao rodar, você verá uma saída no terminal parecida com a do exemplo a seguir:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py makemigrations
medicSearch
Migrations for 'medicSearch':
  medicSearch\migrations\0001_initial.py
```

```
    - Create model Profile
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Agora rodaremos nosso comando de migrate para aplicar as alterações que queremos em nossa base de dados, criando nossa tabela profile.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
medicSearch
```

Teremos uma saída parecida com a que está a seguir:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
medicSearch
Operations to perform:
  Apply all migrations: medicSearch
Running migrations:
  Applying medicSearch.0001_initial... OK
```

Aqui podemos ver que sempre que uma mudança ocorrer em alguma `model` é necessário rodar o comando `makemigrations` e, em seguida, o comando `migrate`.

Nesse exemplo, vimos que ele criou um arquivo chamado `medicSearch.0001_initial.py`. Se abrirmos a pasta `medicSearch\migrations`, dentro dela teremos esse arquivo mais ou menos da seguinte maneira.

medicSearch/migrations/0001_initial.py

```
# Generated by Django 3.1.5 on...
from django.conf import settings
from django.db import migrations, models
import django.db.models.deletion

class Migration(migrations.Migration):
    initial = True

    dependencies = [
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
```

```
    ]

    operations = [
        migrations.CreateModel(
            name='Profile',
            fields=[
                ('id', models.AutoField(auto_created=True,
primary_key=True, serialize=False, verbose_name='ID')),
                ('role', models.IntegerField(choices=[(1, 'Admin'), (2,
'Médico'), (3, 'Paciente')], default=3)),
                ('birthday', models.DateField(blank=True, default=None,
null=True)),
                ('created_at', models.DateTimeField(auto_now_add=True)),
                ('updated_at', models.DateTimeField(auto_now=True)),
                ('token', models.CharField(blank=True, max_length=255,
null=True)),
                ('user',
models.OneToOneField(on_delete=django.db.models.deletion.CASCADE,
to=settings.AUTH_USER_MODEL)),
            ],
        ),
    ]
```

O que o comando `makemigrations` fez foi criar esse arquivo. Ao rodar o `migrate`, realizamos as mudanças criadas nesse arquivo dentro do nosso banco de dados. Se fizermos outra mudança em alguma `model`, rodamos o `makemigrations` novamente e ele criará outro arquivo, depois rodamos novamente o `migrate` para aplicar as mudanças. E fazemos isso sempre que necessário.

Como você percebeu, após o comando `makemigrations` e `migrate`, eu adicionei o nome `medicSearch`. Isso serve para adicionar a ordem em que desejo que ocorra a migração. Quando houver dois ou mais apps criados, você poderá colocar um ao lado do outro, como `python manage.py migrate medicSearch app3 app2`, por exemplo. Isso é necessário, pois se você criar uma nova `model` no app3 e adicionar um relacionamento entre essa `model` e uma já existente do app2, por exemplo, ao realizar o `migrate`, antes de criar a tabela do app3 teremos um erro de migração.

Para entrar no admin, acesse http://127.0.0.1:8080/admin. Adicione o usuário e senha que criamos e pronto, você estará no painel admin.

Não se esqueça de que a aplicação precisa estar rodando. Se você a tiver parado para rodar a migração, rode o comando `runserver` novamente (python manage.py runserver 127.0.0.1:8080).

Você notará que ainda não veremos o painel de perfil, mas fique tranquilo que veremos como resolver isso a seguir:

Figura 3.1: Painel admin

Isso acontece porque ainda não falamos para o Django que queremos exibir nossa tabela de perfil no painel admin. Precisamos abrir o arquivo `admin.py` que está dentro do diretório `medicSearch` e adicionar o código a seguir:

medicSearch/admin.py

```
from django.contrib import admin
from .models import *

admin.site.register(Profile)
```

O método `admin.site.register(Profile)` permite que exibamos nossa model no painel admin. Veja:

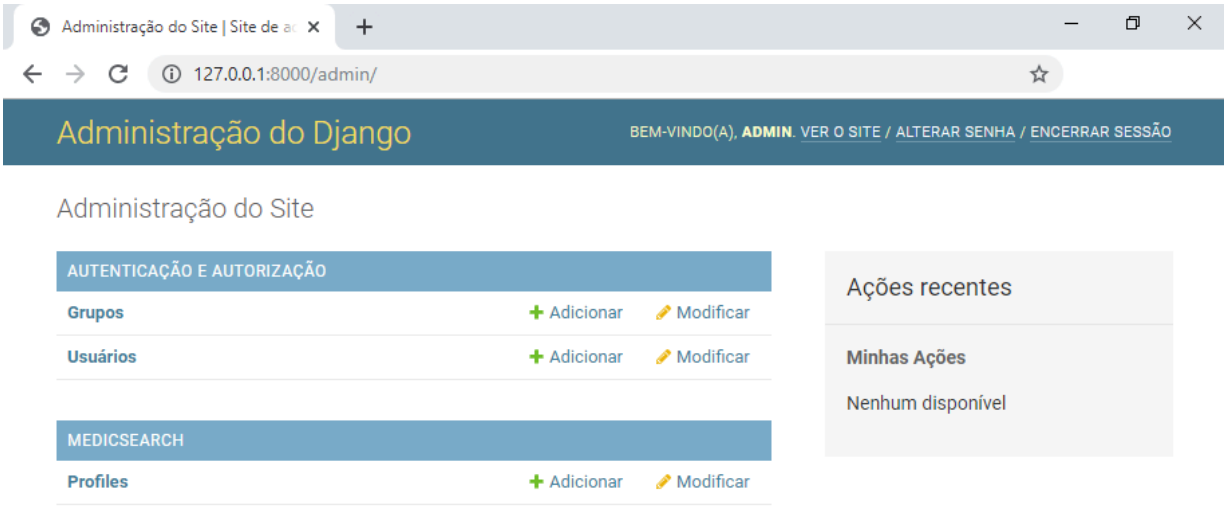


Figura 3.2: Painel admin

Se criarmos um novo usuário, automaticamente um perfil será criado para ele. Não é necessário criar um perfil para o usuário toda vez que um novo usuário for criado. Veja só:

Tela de usuário com novo usuário:

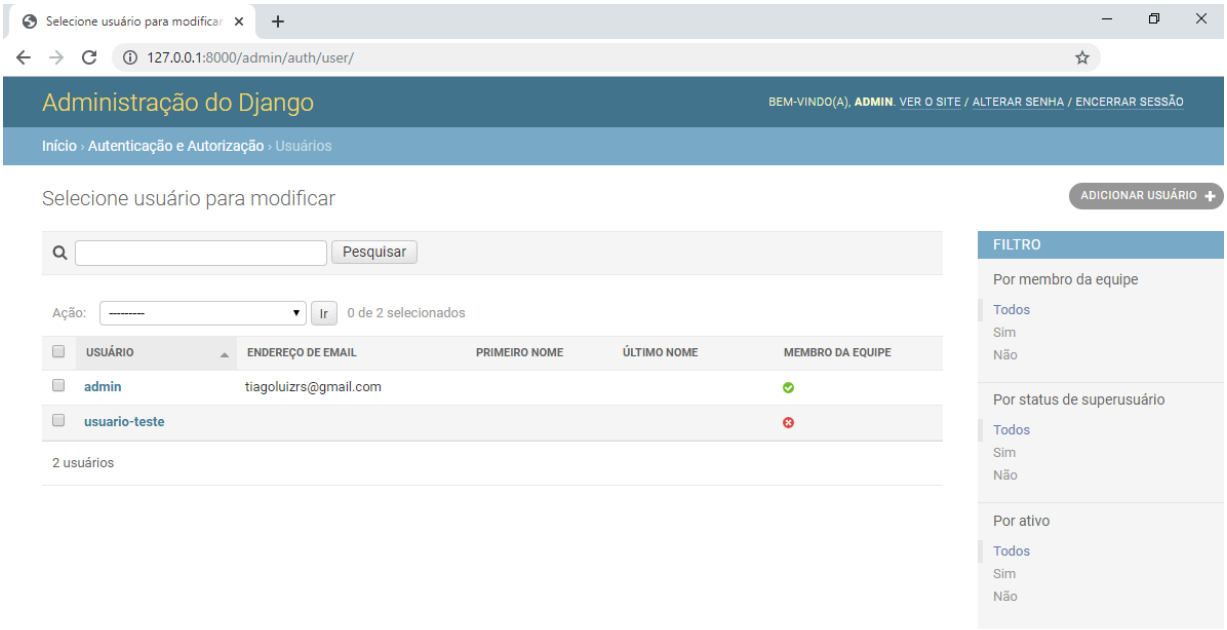


Figura 3.3: Painel admin - Usuário

Tela de perfil com perfil do usuário já criado:

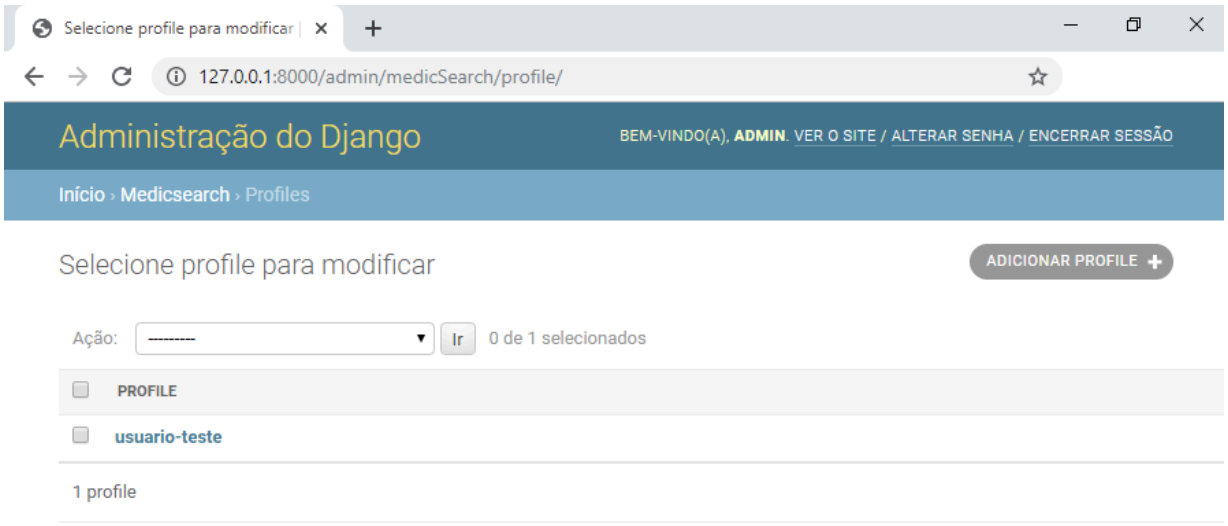


Figura 3.4: Painel admin - Perfil

Perceba que na tela de perfis o usuário admin não tem perfil. Isso ocorre porque ele foi criado antes de a tabela perfil existir. Entre na tela de perfis e crie um perfil para o usuário admin.

Criando perfil:

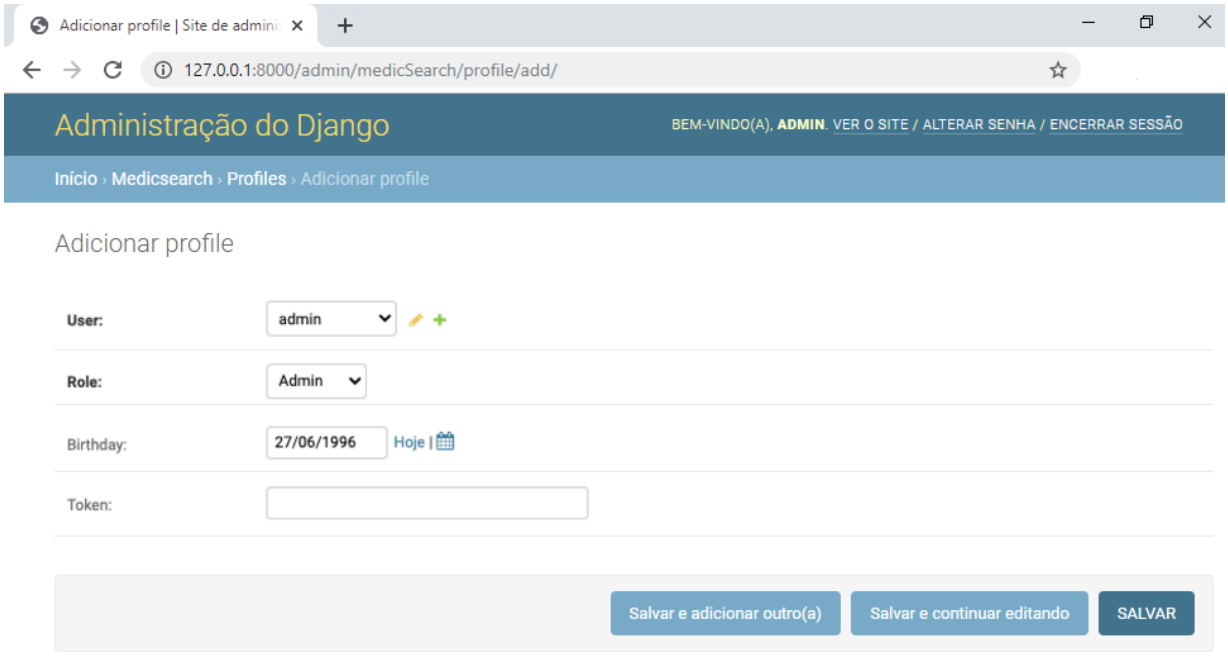


Figura 3.5: Painel admin - Novo perfil para o usuário admin

Perfil criado:

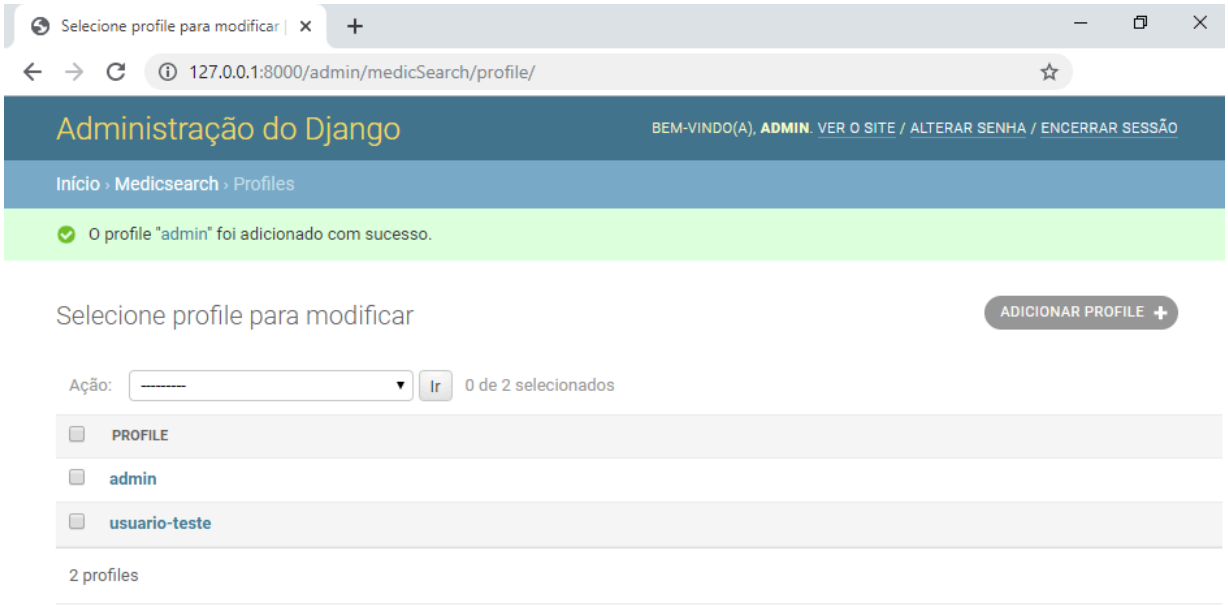


Figura 3.6: Painel admin - Perfil do usuário admin criado

Fique atento: o nome do usuário é admin, mas poderia ser qualquer outro. O fato aqui é que o primeiro usuário criado não possui perfil e precisa receber um.

É possível customizarmos essa visualização, mas por enquanto não faremos isso.

A tabela perfil aqui é só para gerenciar os acessos de api e para o nosso futuro front-end customizado. O painel admin é gerenciado através da tabela de usuários. No capítulo sobre painel administrativo, vamos entender um pouco sobre a tela de usuários do painel admin.

3.4 Criando e customizando as models restantes

Nesta etapa, vamos criar as primeiras models do nosso sistema, inclusive modificaremos algumas coisas no perfil do nosso usuário também.

As models que criaremos agora serão:

- Speciality: especialidades que serão atribuídas aos médicos.
- DayWeek: dias da semana em que abrirá.
- State: estados do país.
- City: cidades de um estado.
- Neighborhood: bairros de uma cidade.
- Address: endereços em que o médico atende.
- Rating: tabela onde serão adicionadas as pontuações aos médicos.

Vamos lá. Dentro do diretório `models` crie o arquivo `Speciality.py` e vamos adicionar o seguinte código a ele:

medicSearch/models/Speciality.py

```
from medicSearch.models import *

class Speciality(models.Model):
    name = models.CharField(null=False, max_length=100)
    status = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{}'.format(self.name)
```

Vamos criar a model de dias da semana. Crie um arquivo chamado `DayWeek.py` no diretório `models`.

medicSearch/models/DayWeek.py

```
from medicSearch.models import *

class DayWeek(models.Model):
    name = models.CharField(null=False, max_length=20)
    status = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{}'.format(self.name)
```

Agora vamos criar a model de estados. Crie um arquivo chamado `State.py` no diretório `models`.

medicSearch/models/State.py

```
from medicSearch.models import *

class State(models.Model):
    name = models.CharField(null=False, max_length=20)
    status = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

```
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{}'.format(self.name)
```

A seguir, vamos criar a model de cidades. Crie um arquivo chamado `City.py` no diretório `models`.

medicSearch/models/City.py

```
from medicSearch.models import *

class City(models.Model):
    state = models.ForeignKey(State, null=True, related_name='state',
on_delete=models.SET_NULL)
    name = models.CharField(null=False, max_length=20)
    status = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{} - {}'.format(self.name, self.state.name)
```

Vamos agora criar a model de bairros. Crie um arquivo chamado `Neighborhood.py` no diretório `models`.

medicSearch/models/Neighborhood.py

```
from medicSearch.models import *

class Neighborhood(models.Model):
    city = models.ForeignKey(City, null=True, related_name='city',
on_delete=models.SET_NULL)
    name = models.CharField(null=False, max_length=20)
    status = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{} - {}'.format(self.name, self.city.name)
```

Agora que criamos a model de especialidades, estados, cidades, bairros e de dias da semana, vamos criar a model de endereços do médico. Será um novo arquivo dentro do diretório `models`, chamado `Address.py` e adicionaremos a ele o seguinte código:

medicSearch/models/Address.py

```
from medicSearch.models import *

class Address(models.Model):
    neighborhood = models.ForeignKey(Neighborhood, null=True,
related_name='neighborhood', on_delete=models.SET_NULL)
    name = models.CharField(null=False, max_length=100)
    address = models.CharField(null=False, max_length=255)
    latitude = models.DecimalField(max_digits=9, decimal_places=7)
    longitude = models.DecimalField(max_digits=9, decimal_places= 7)
    opening_time = models.TimeField()
    closing_time = models.TimeField()
    days_week = models.ManyToManyField(DayWeek, blank=True,
related_name='days_week')
    phone = models.CharField(null=True, blank=True, max_length=50)
    status = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return '{}'.format(self.name)
```

Para fechar as criações, vamos criar o arquivo `Rating.py` dentro do diretório `models`.

medicSearch/models/Rating.py

```
from medicSearch.models import *

class Rating(models.Model):
    user = models.ForeignKey(User, related_name='avaliou',
on_delete=models.CASCADE)
    user_rated = models.ForeignKey(User, related_name='avaliado',
on_delete=models.CASCADE)
```

```
    value = models.DecimalField(max_digits=5, decimal_places=2)
    opinion = models.TextField(null=True, blank=True)
    status = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True, null=True)
    updated_at = models.DateTimeField(auto_now=True, null=True)

    def __str__(self):
        return 'Avaliador: {} | Avaliado: {}'.format(self.user.first_name,
self.user_rated)
```

Precisamos abrir nosso arquivo `__init__.py` dentro do diretório de models e importar as novas models.

medicSearch/models/__init__.py

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

ROLE_CHOICE = (
    (1, 'Admin'),
    (2, 'Médico'),
    (3, 'Paciente')
)

from .Rating import Rating
from .DayWeek import DayWeek
from .State import State
from .City import City
from .Neighborhood import Neighborhood
from .Address import Address
from .Speciality import Speciality
from .Profile import Profile
```

Antes de rodarmos nossa aplicação, precisamos fazer uma alteração na model de perfil, adicionando a ela três campos de `ManyToMany` para que o médico possa adicionar em seu perfil os endereços onde ele atenderá e suas especialidades, bem como para que o paciente possa adicionar em seu perfil os médicos favoritos.

medicSearch/models/Profile.py

```
from medicSearch.models import *

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    role = models.IntegerField(choices=ROLE_CHOICE, default=3)
    birthday = models.DateField(default=None, null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    token = models.CharField(max_length=255, null=True, blank=True)
    # Adicione as linhas a seguir no seu arquivo `Profile.py`
    favorites = models.ManyToManyField(User, blank=True,
related_name='favorites')
    specialties = models.ManyToManyField(Speciality, blank=True,
related_name='specialties')
    addresses = models.ManyToManyField(Address, blank=True,
related_name='addresses')
```

Para que as alterações sejam aplicadas, precisaremos rodar novamente os comandos `makemigrations` e `migrate`.

1. Makemigrations

   ```
   (venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py
   makemigrations medicSearch
   ```

2. Migrate

   ```
   (venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
   medicSearch
   ```

Assim como fizemos com o Profile, precisamos adicionar as novas models ao arquivo `admin.py`, então abra-o e altere conforme o código a seguir:

medicSearch/admin.py

```
from django.contrib import admin
from .models import *
```

```
admin.site.register(Profile)
admin.site.register(State)
admin.site.register(City)
admin.site.register(Neighborhood)
admin.site.register(Address)
admin.site.register(DayWeek)
admin.site.register(Rating)
admin.site.register(Speciality)
```

Agora conseguiremos ver nossas novas models no sistema.

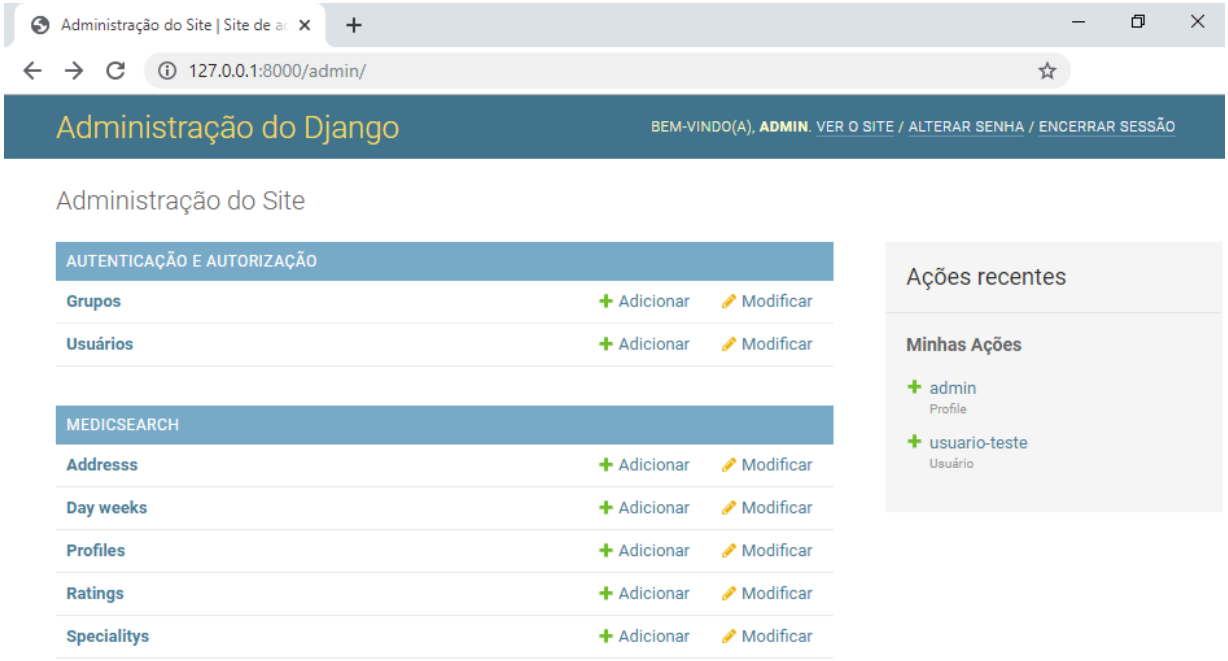


Figura 3.7: Painel admin - Novas models

3.5 Fluxo de criação de um usuário no admin

Mais à frente veremos como será feita a criação de um perfil de paciente e de médico pelo front-end que criaremos. Quando isso

ocorrer, já teremos especialidades criadas e também uma tela para o usuário criar seus locais de trabalho, mas por enquanto já podemos fazer isso através do admin, e para isso temos os fluxos a seguir.

No caso da criação de um médico, mesmo que não haja uma especialidade ou um local de trabalho (Address) podemos fazer a criação antes de entrar no painel de Profile ou até mesmo dentro do painel de Profile.

Vemos a seguir um fluxo:

1. Criação de um dia da semana:

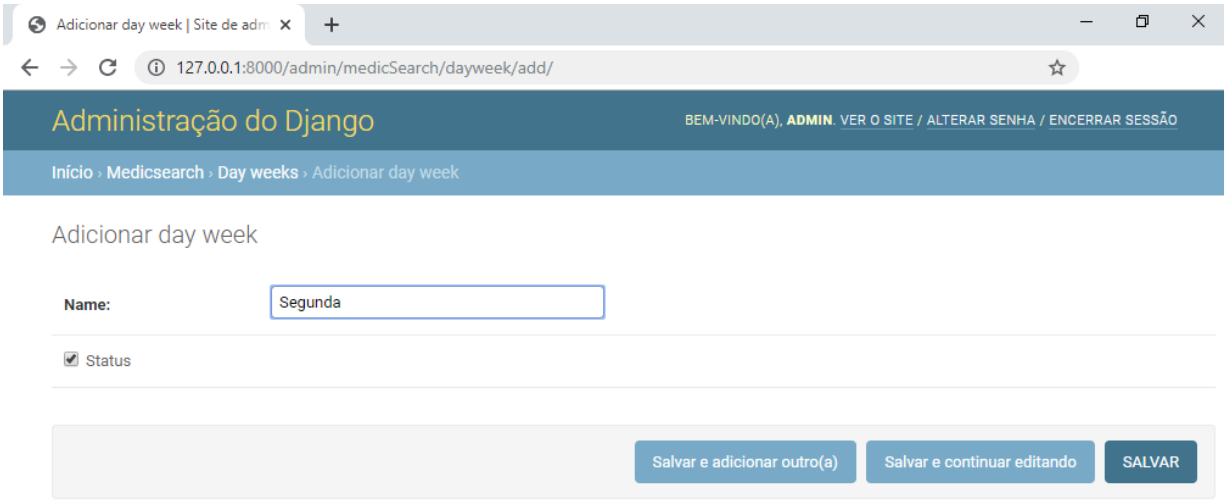


Figura 3.8: Painel admin - Novo dia da semana

2. Criação de um estado:

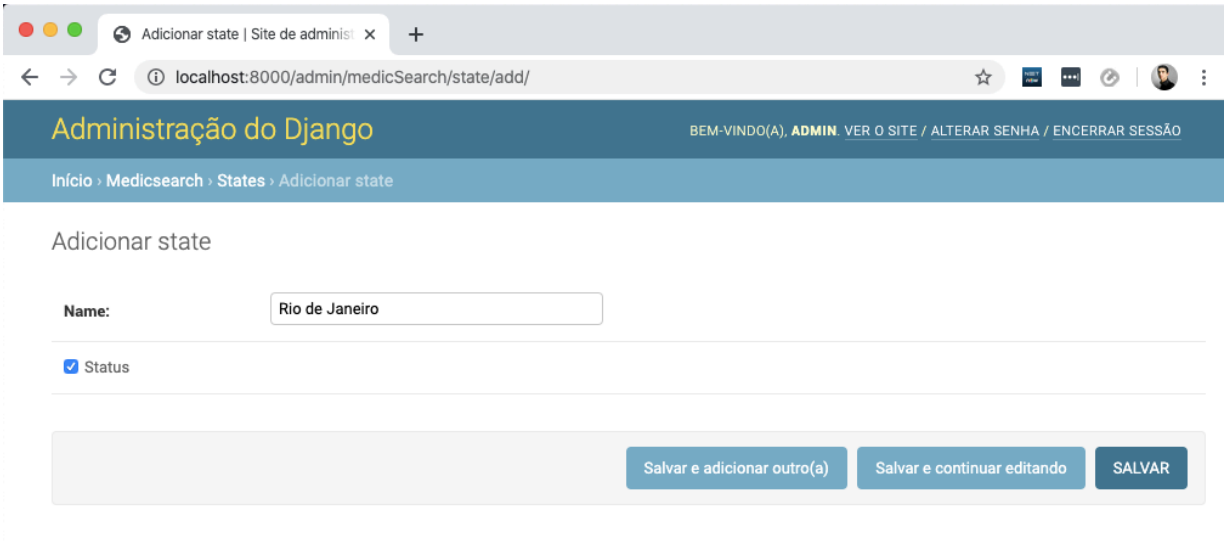


Figura 3.9: Painel admin - Novo estado

3. Criação de uma cidade:

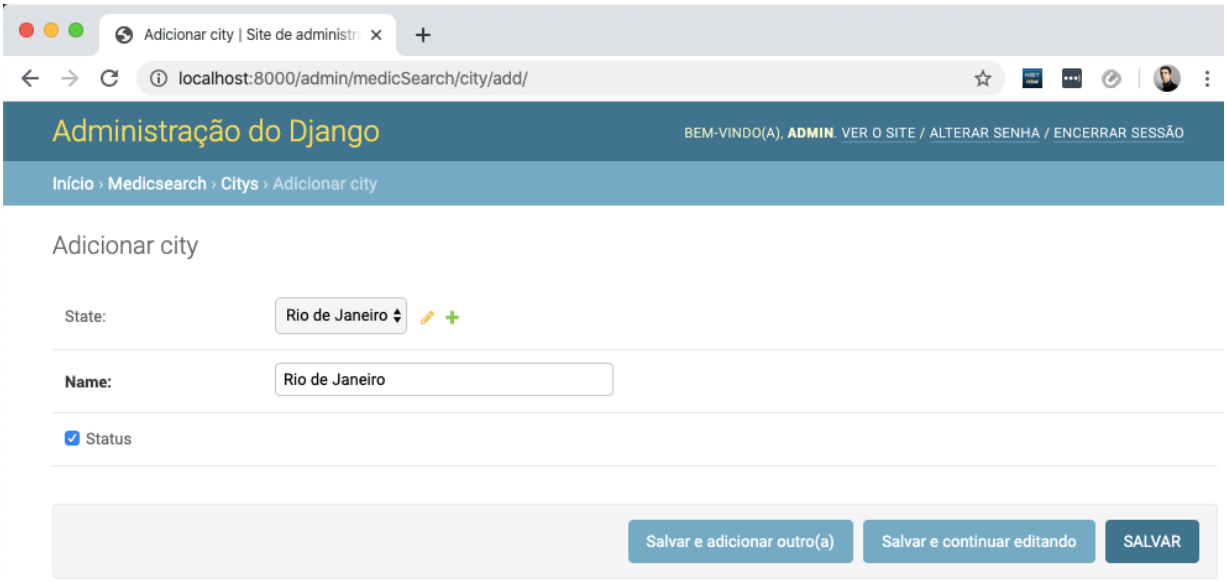


Figura 3.10: Painel admin - Nova cidade

4. Criação de um bairro:

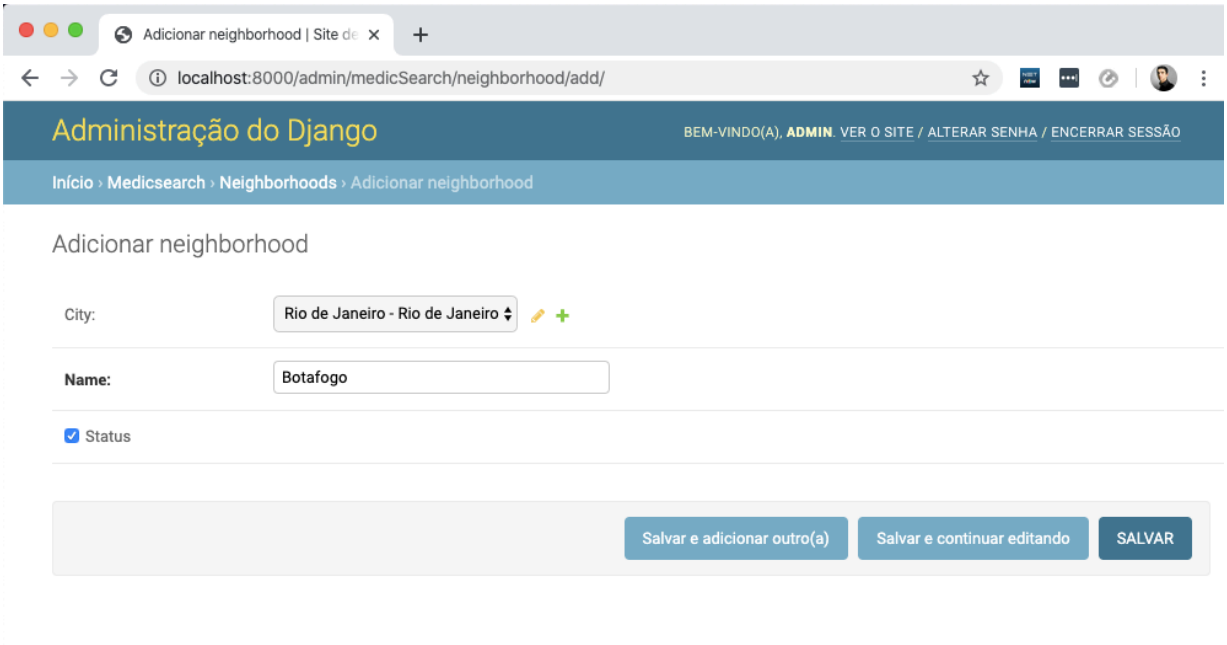


Figura 3.11: Painel admin - Novo bairro

5. Criação de um local de atendimento:

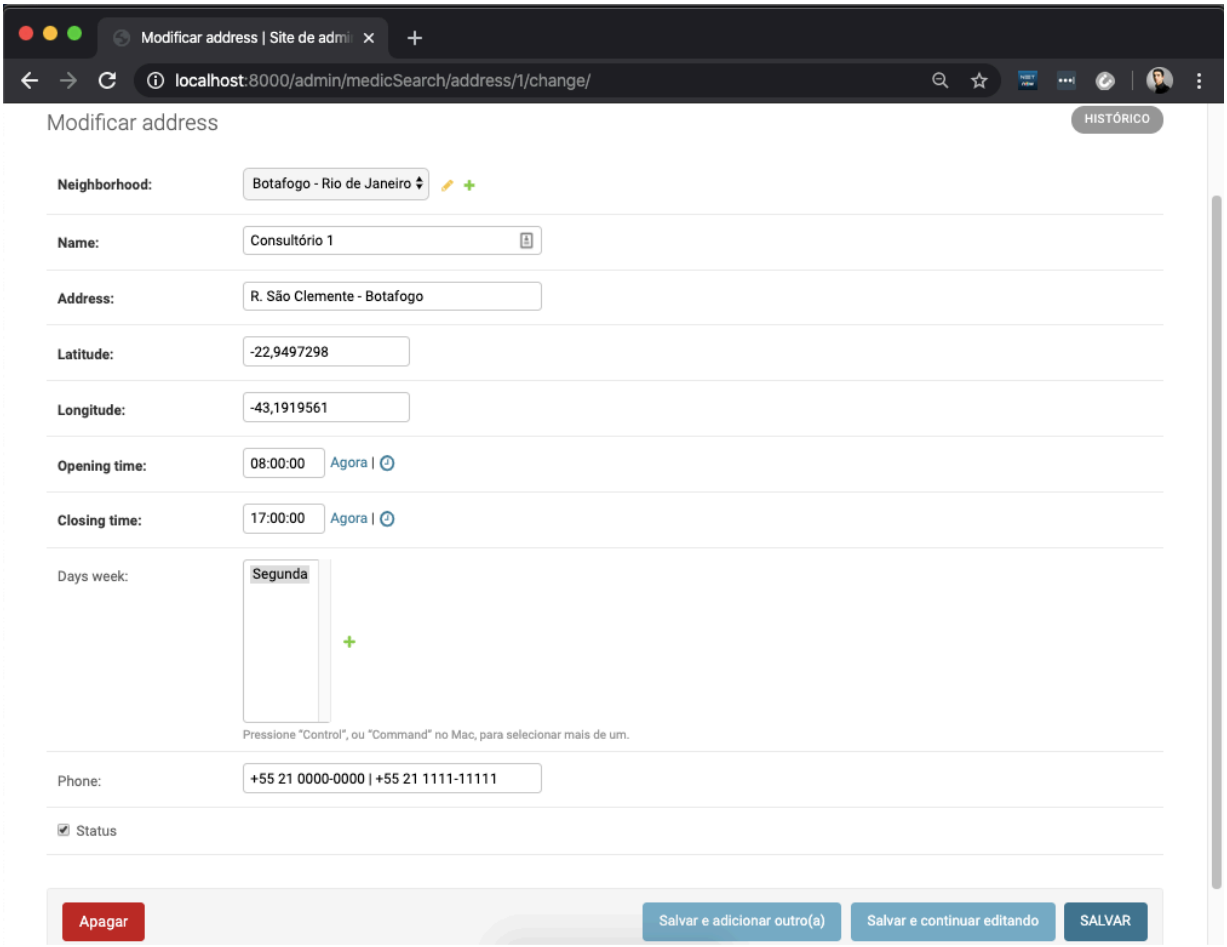


Figura 3.12: Painel admin - Novo local de atendimento

Você pode usar o site https://www.latlong.net/convert-address-to-lat-long.html para pegar a latitude e longitude de um endereço.

6. Criação de uma especialidade:

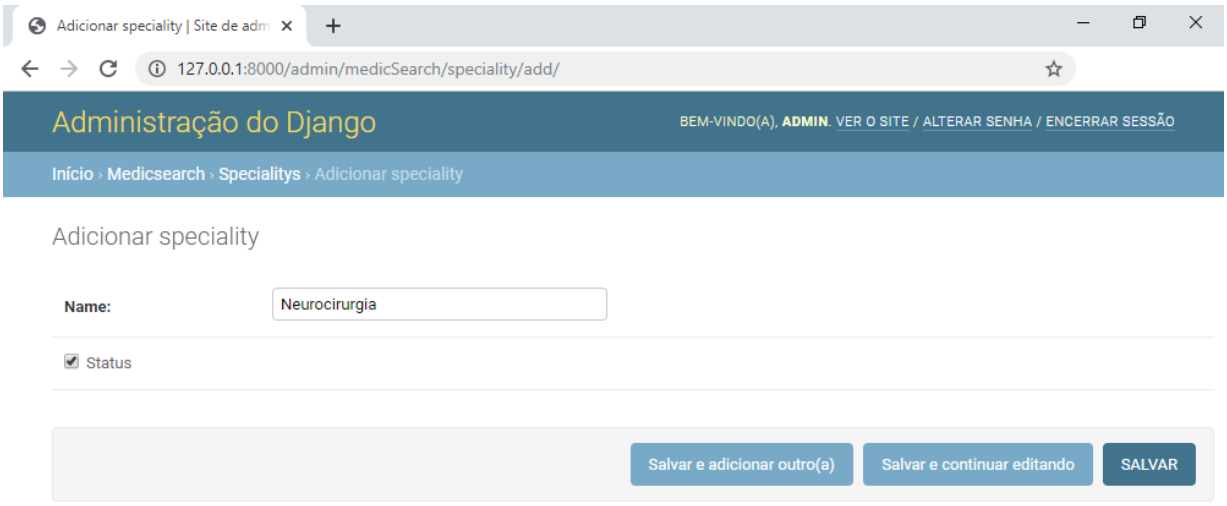


Figura 3.13: Painel admin - Nova especialidade

7. Alteração do perfil do médico para adicionar os itens criados agora:

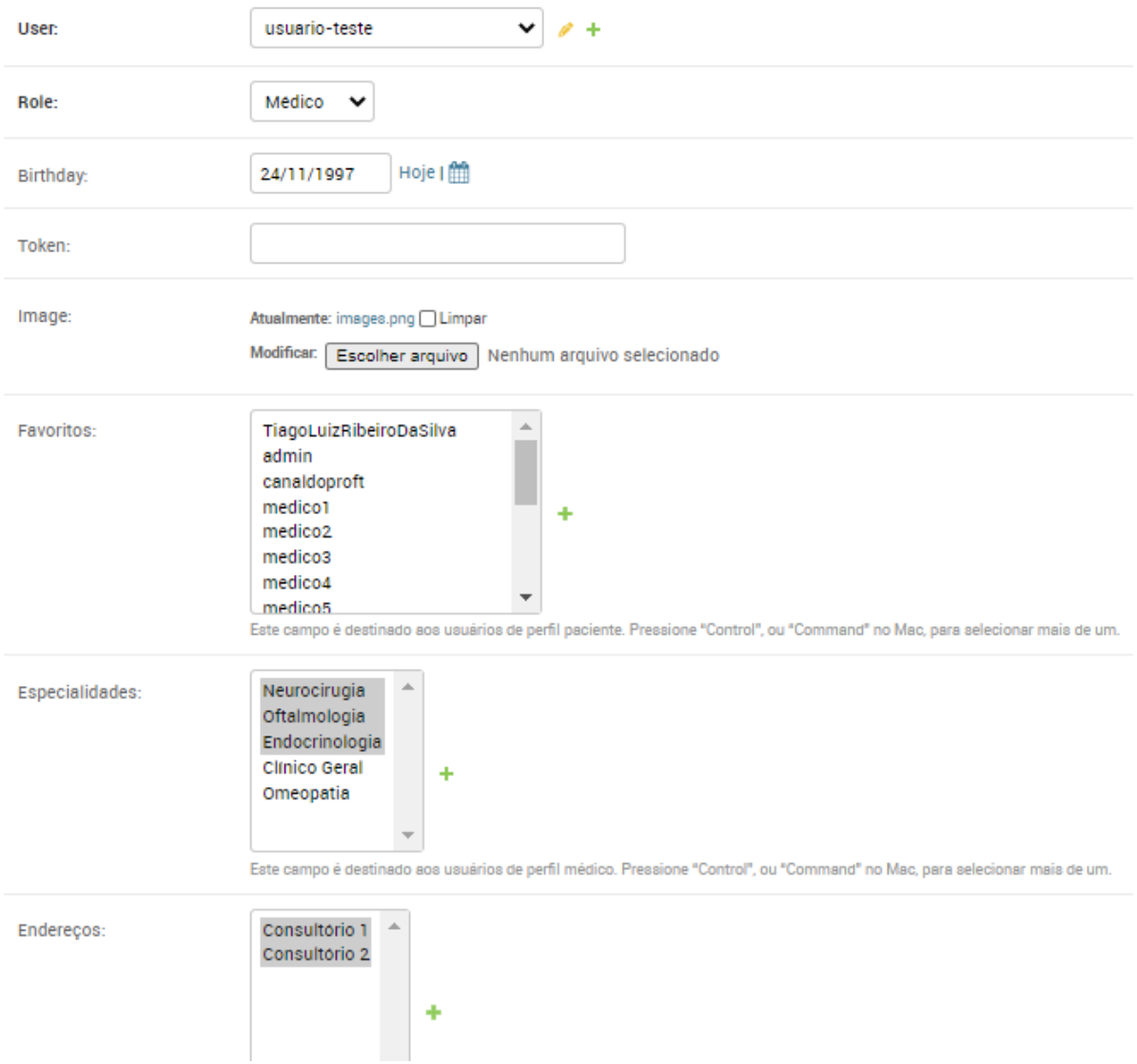


Figura 3.14: Painel admin - Edição do perfil para médico

Posteriormente, veremos a criação de uma avaliação. Por ora foi possível ver que as models às quais foram adicionados relacionamentos `ManyToMany`, `OneToOne` e `ForeingKey` possuem campos dinâmicos.

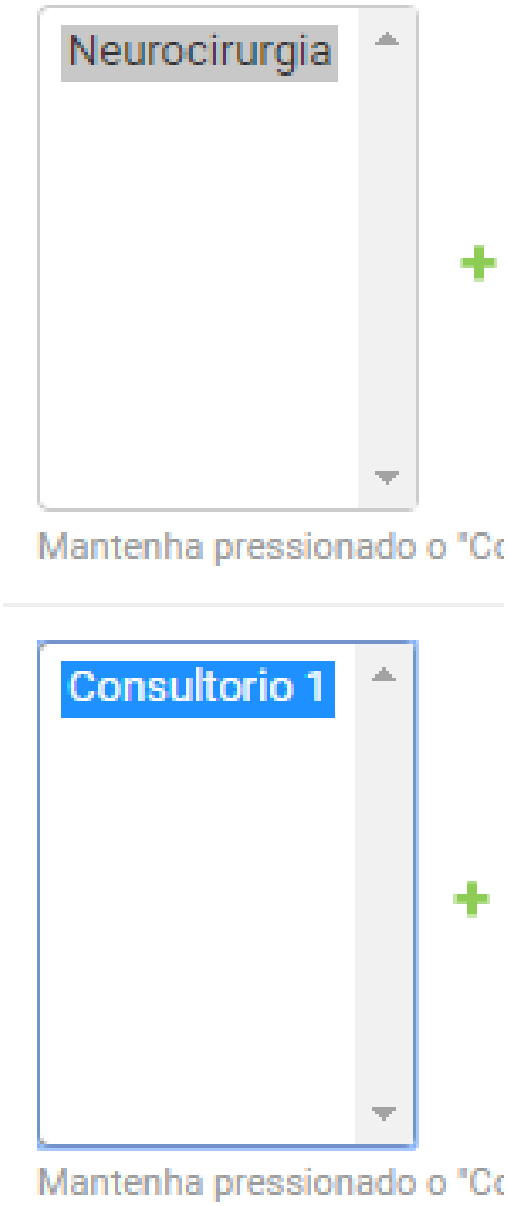


Figura 3.15: Painel admin - Relacionamento dentro da model

Todo esse dinamismo ocorre pela configuração que fizemos em cada model que criamos em nosso projeto.

3.6 Upload de imagens

Nesta seção, vamos ver como fazer upload de imagens em nossa plataforma. Para isso é necessário instalarmos a biblioteca `Pillow`, responsável por salvar a imagem no sistema. Não vamos usá-la diretamente, quem fará isso será o Django, mas precisamos instalá-la para que ele consiga usá-la. Vamos lá.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install Pillow
```

Agora que a biblioteca foi instalada, vamos adicionar uma linha de código em nossa model `Profile.py`. Essa linha será um atributo chamado `image`, que receberá o objeto `ImageField`, responsável por criar um campo de upload de imagem no sistema.

medicSearch/models/Profile.py

```
from medicSearch.models import *

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    role = models.IntegerField(choices=ROLE_CHOICE, default=3)
    birthday = models.DateField(default=None, null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    token = models.CharField(max_length=255, null=True, blank=True)
    # Adicione as linhas a seguir no seu arquivo `Profile.py`
    image = models.ImageField(null=True, blank=True)
    # Adicione antes das três linhas abaixo para deixar organizado
    favorites = models.ManyToManyField(User, null=True, blank=True,
related_name='favorites')
    specialties = models.ManyToManyField(Speciality, null=True,
blank=True, related_name='specialties')
    addresses = models.ManyToManyField(Address, null=True, blank=True,
related_name='addresses')
```

Com isso feito, precisamos dizer ao Django qual será o local em que salvaremos nossos arquivos. Para isso, abra o arquivo `settings.py` dentro da pasta `medicSearchAdmin/settings`. Nele, colocaremos as constantes `MEDIA_URL` e `MEDIA_ROOT`.

medicSearchAdmin/settings/settings.py

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')

# Adicione as linhas a seguir após as linhas STATIC_URL e STATIC_ROOT
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

Vamos entender para que cada uma delas serve.

- **MEDIA_ROOT**: caminho absoluto do sistema de arquivos para o diretório que conterá os arquivos enviados pelo usuário.
  `Exemplo: "/var/www/example.com/media/"` .
- **MEDIA_URL**: URL que lida com a mídia veiculada `MEDIA_ROOT` , usada para gerenciar arquivos armazenados. Exemplo:
  `"http://media.example.com/"` .

Após isso, vamos abrir o arquivo `urls.py` que está na pasta `medicSearchAdmin` e alterar conforme o exemplo a seguir:

medicSearchAdmin/urls.py

```
from django.contrib import admin
from django.urls import path
# Adicione os imports a seguir
from django.conf.urls.static import static
from django.conf import settings

# Altere a linha a seguir adicionando o + e todos os itens que vem a
seguir dele.
urlpatterns = [
    path('admin/', admin.site.urls),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT) +
static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Aqui estamos dizendo ao Django que dentro do admin poderemos acessar o diretório de imagens (media) e o static (falaremos dele mais à frente).

Para fechar, vamos rodar o `makemigrations` e o `migrate` para que seja adicionada a nova coluna `image` em nosso banco de dados:

1. Makemigrations

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py
makemigrations medicSearch
```

2. Migrate

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
medicSearch
```

Após o `migrate` tudo está certo e já podemos adicionar imagens em nosso perfil. Veja a imagem a seguir:

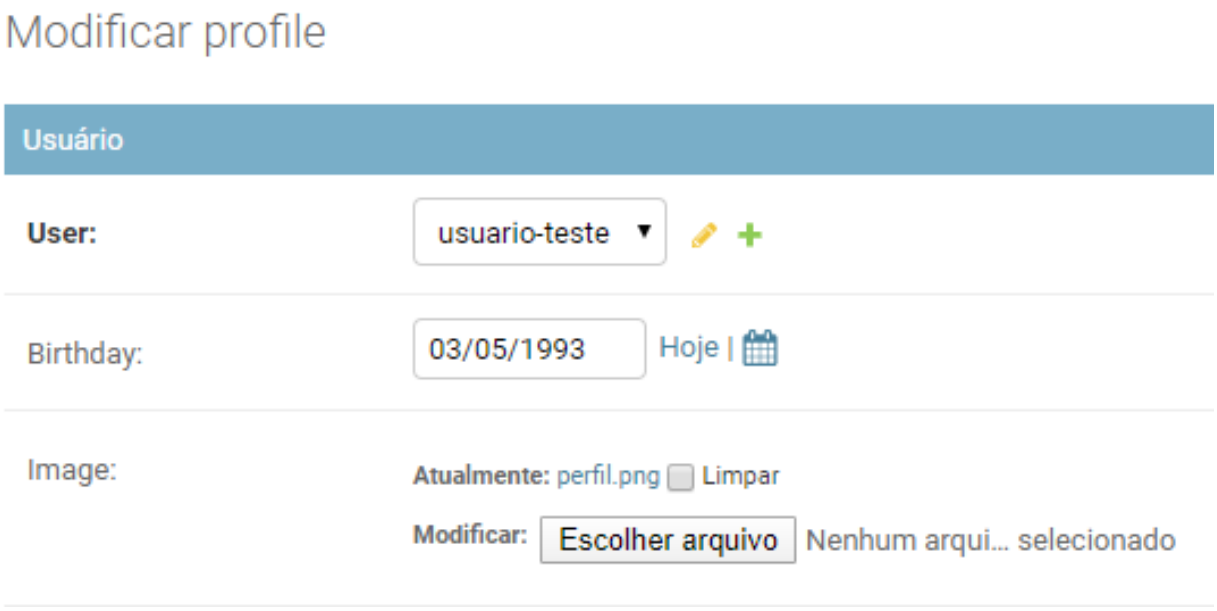


Figura 3.16: Upload de imagem no perfil

Se você editar o perfil, após salvar, ele mostrará um link azul, e clicando nele a imagem abrirá na url do seu sistema, `http://127.0.0.1:8000/media/perfil.png` .

Fique atento! No exemplo anterior fizemos um upload de uma imagem que se chamava `perfil.png`.

Neste capítulo, aprendemos a configurar as models do nosso projeto de modo que consigamos realizar relacionamentos entre tabelas e ter uma estrutura de dados bem consistente. No próximo capítulo, aprenderemos a personalizar nosso painel administrativo.

Deixo aqui a estrutura principal do projeto finalizada até o presente momento:

```
livro_django
├──medicSearch
│        migrations
│        models
│        │     __init__.py
│        │     Address.py
│        │     City.py
│        │     DayWeek.py
│        │     Neighborhood.py
│        │     Profile.py
│        │     Rating.py
│        │     Speciality.py
│        │     State.py
│        │
│        __init__.py
│        admin.py
│        apps.py
│        tests.py
│        views.py
├──medicSearchAdmin
│        media
│        settings
│        │       settings.py
│        │       development.py
│        │       production.py
│        │       testing.py
│        __init__.py
```

```
│        db.sqlite3
│        urls.py
│        wsgi.py
├──venv
├──manage.py
└──requirements.txt
```

Capítulo 4
Área administrativa

Como vimos no capítulo anterior, o Django nos proporciona uma área administrativa, onde é possível gerenciarmos os dados que persistirem em nosso banco. Inicialmente, o admin pode parecer muito vazio, apenas expondo os dados através do conhecido CRUD (CREATE, READ, UPDATE e DELETE), mas existem diversas outras coisas que podemos fazer no painel administrativo, como **filtros**, **pesquisas**, **emissão de relatórios** entre muitos outros itens que veremos no decorrer deste capítulo. Veremos como personalizar nosso painel admin ou, como chamamos no Django, nossa *model admin* para que ela possa ser bem mais do que um painel, no qual vamos inserir e gerenciar dados do nosso sistema.

4.1 Customizando o admin

Nesta etapa, nós aprenderemos a customizar nosso admin com alguns componentes muito interessantes que permitirão alterar nossa forma de listar os dados, a aparência dos formulários e também adicionar arquivos `csv` e `js` em nosso projeto. Vamos lá.

No capítulo anterior, aprendemos a registrar nossas models no painel administrativo para que elas sejam exibidas em nosso sistema e assim possamos gerenciar os dados que correspondem a cada model no banco de dados. A seguir, veremos um modelo de criação de model customizada e para isso customizaremos a model admin do `Profile`. Altere o arquivo `admin.py` que está na pasta `medicSearch`.

medicSearch/admin.py

```
from django.contrib import admin
from .models import *
```

```
# Crie o método a seguir em seu arquivo
class ProfileAdmin(admin.ModelAdmin):
    # Cria um filtro de hierarquia com datas
    date_hierarchy = 'created_at'

# Agora altere o register da model Profile.
admin.site.register(Profile, ProfileAdmin)
admin.site.register(Address)
admin.site.register(DayWeek)
admin.site.register(Rating)
admin.site.register(Speciality)
```

Como vimos no exemplo, foi criada uma classe chamada `ProfileAdmin`, dentro da qual podemos adicionar atributos que serão responsáveis por alterar a aparência e o comportamento do nosso admin. Veja que, além da classe que criamos, foi necessário passar essa classe `ProfileAdmin` como um parâmetro do método `register`. Será nesse momento que registraremos essas nossas configurações customizadas no admin daquela model específica - em nosso caso, da model `Profile`.

Vamos conhecer alguns componentes que nos permitirão customizar o admin. Todos eles devem ser colocados um abaixo do outro. No exemplo anterior, temos o `date_hierarchy` que nos permite separar os dados da model `Profile` pela data de criação. Para adicionar os componentes que veremos a seguir, basta colocá-los abaixo do componente `date_hierarchy`.

date_hierarchy: esse componente nos permite criar um filtro de data dentro da listagem do admin. O campo a ser passado nele precisa ser de tipo `DateField`, caso contrário não funcionará:

```
class ProfileAdmin(admin.ModelAdmin):
    date_hierarchy = 'created_at'

admin.site.register(Profile, ProfileAdmin)
```

Figura 4.1: Customização do admin - Date Hierarchy

list_display: esse componente nos permite dizer ao Django quais elementos desejamos que sejam exibidos na listagem dos dados da model no admin:

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'role', 'birthday',)

admin.site.register(Profile, ProfileAdmin)
```

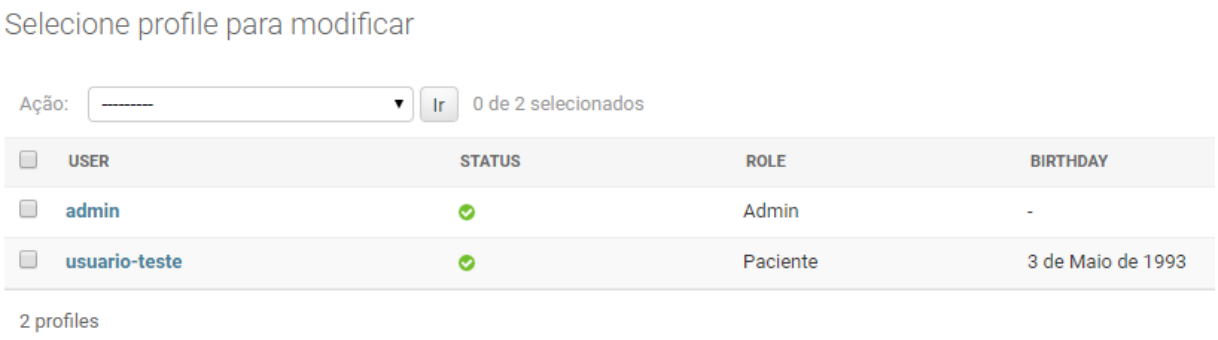


Figura 4.2: Customização do admin - List Display

Perceba que ainda não adicionamos os campos de tipo `Foreing Key`, `OneToMany` ou `ManyToMany`. Existem algumas particularidades no `list_display` para campos que representam relacionamento. Veremos mais à frente.

empty_value_display: esse componente nos permite alterar a apresentação dos campos vazios em nosso sistema:

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'role', 'birth', 'specialtiesList',
'addressesList',)
    empty_value_display = 'Vazio'

admin.site.register(Profile, ProfileAdmin)
```

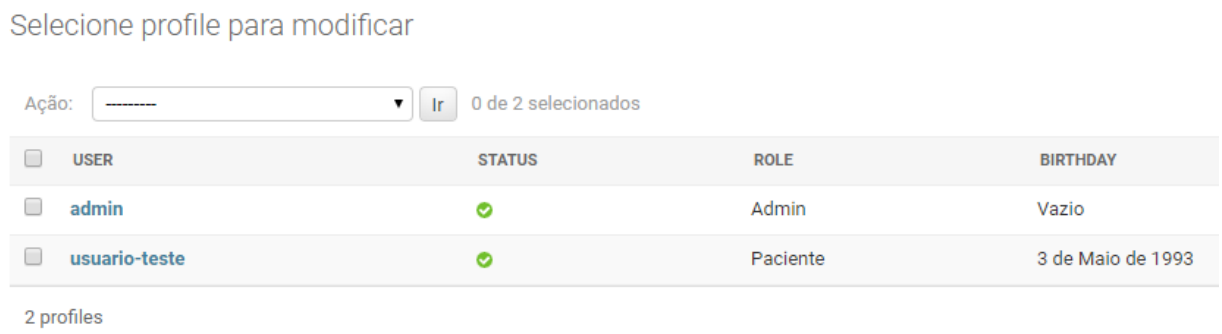


Figura 4.3: Customização do admin - Empty Value Display

> Perceba que deixei o `list_display` para que possamos ver os possíveis campos vazios na model de `Profile`, mas ele não é obrigatório para que o `empty_value_display` seja adicionado.

list_display_links: como percebemos, para clicarmos em um item da listagem, o link para edição do item sempre ficará na primeira coluna do painel. No caso do `Profile`, é a coluna USER. Para podermos deixar outras linhas com o link de edição habilitado, precisamos do `list_display_links`.

> Fique atento a um ponto. Se adicionarmos uma nova coluna como `list_display_links`, ela só funcionará se estiver listada na propriedade `list_display`, aquela que é responsável por exibir as colunas no painel admin. Veja o exemplo a seguir:

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'role', 'birthday',)
    list_display_links = ('user', 'role',)

admin.site.register(Profile, ProfileAdmin)
```

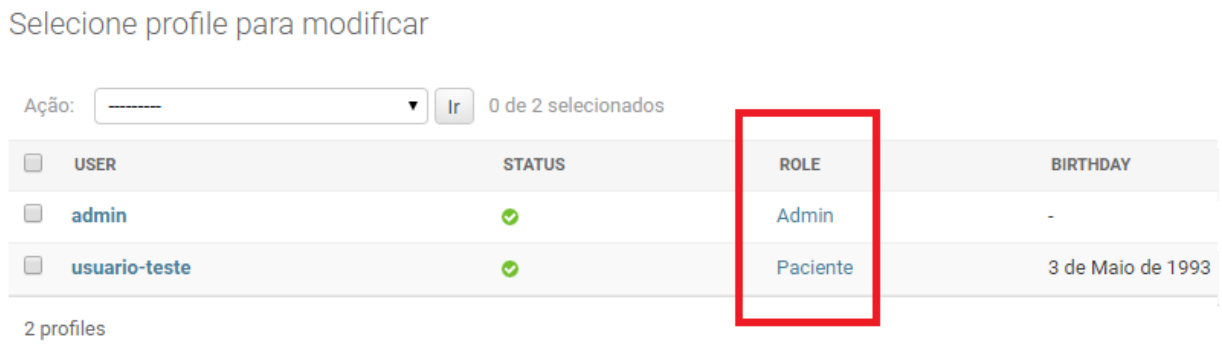


Figura 4.4: Customização do admin - List Display Links

list_filter: permite que criemos um filtro de dados baseado nos campos que forem adicionados a essa lista. Esse filtro fica alocado ao lado direito da tela de listagem da model admin:

```
class ProfileAdmin(admin.ModelAdmin):
    list_filter = ('user__is_active',)

admin.site.register(Profile, ProfileAdmin)
```

Perceba que para acessar um atributo da Foreign Key `user` que faz referência a `model` de usuário, foi necessário chamar o atributo `user` e adicionar dois underlines para separá-lo do atributo que faz referência à model `User`, ficando assim: `user__is_active`.

Figura 4.5: Customização do admin - List Filter

fields: permite dizer quais campos serão exibidos no formulário e quais não serão. Fique atento, pois um campo que seja `not null` sem valor default deverá ser adicionado na listagem, senão um erro ocorrerá na hora de salvar o dado, dizendo que o campo `not null` não pode ficar em branco:

```
class ProfileAdmin(admin.ModelAdmin):
    fields = ('user', ('role', ), 'image', 'birthday', 'specialties',
'addresses',)

admin.site.register(Profile, ProfileAdmin)
```

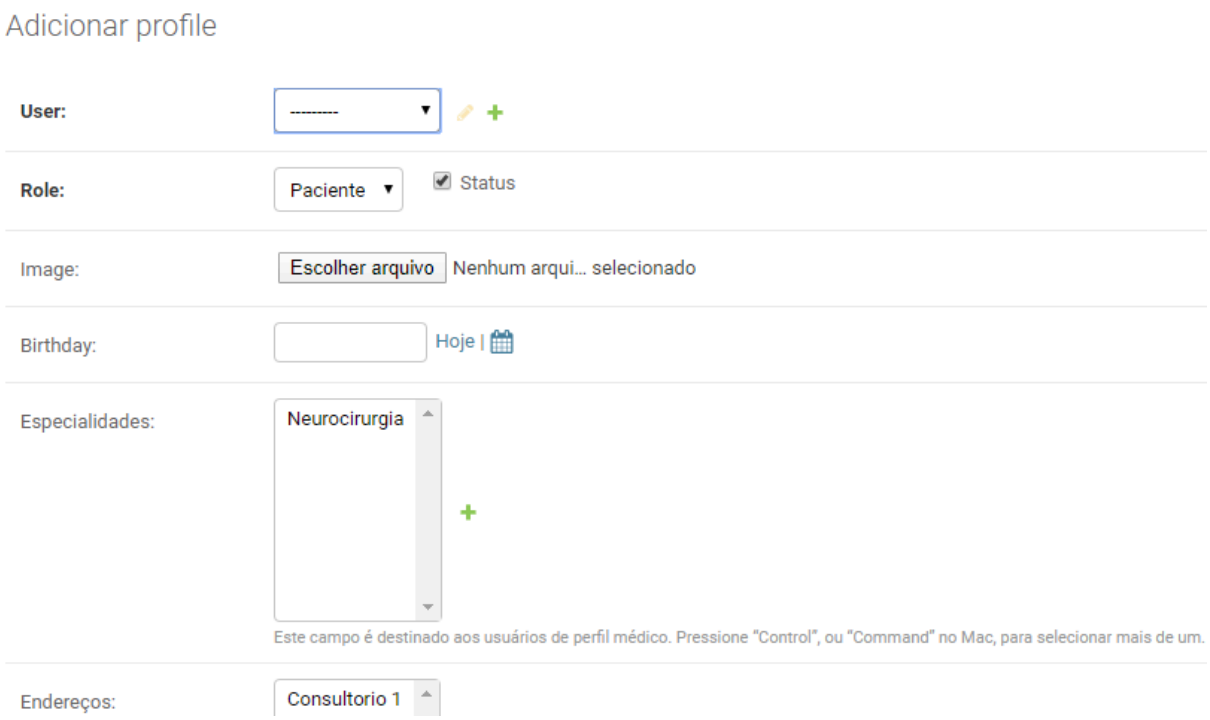


Figura 4.6: Customização do admin - Fields

exclude: é o oposto do `fields`. Ele removerá do formulário os campos que forem adicionados em sua lista:

```
class ProfileAdmin(admin.ModelAdmin):
    exclude = ('favorites', 'created_at', 'updated_at',)

admin.site.register(Profile, ProfileAdmin)
```

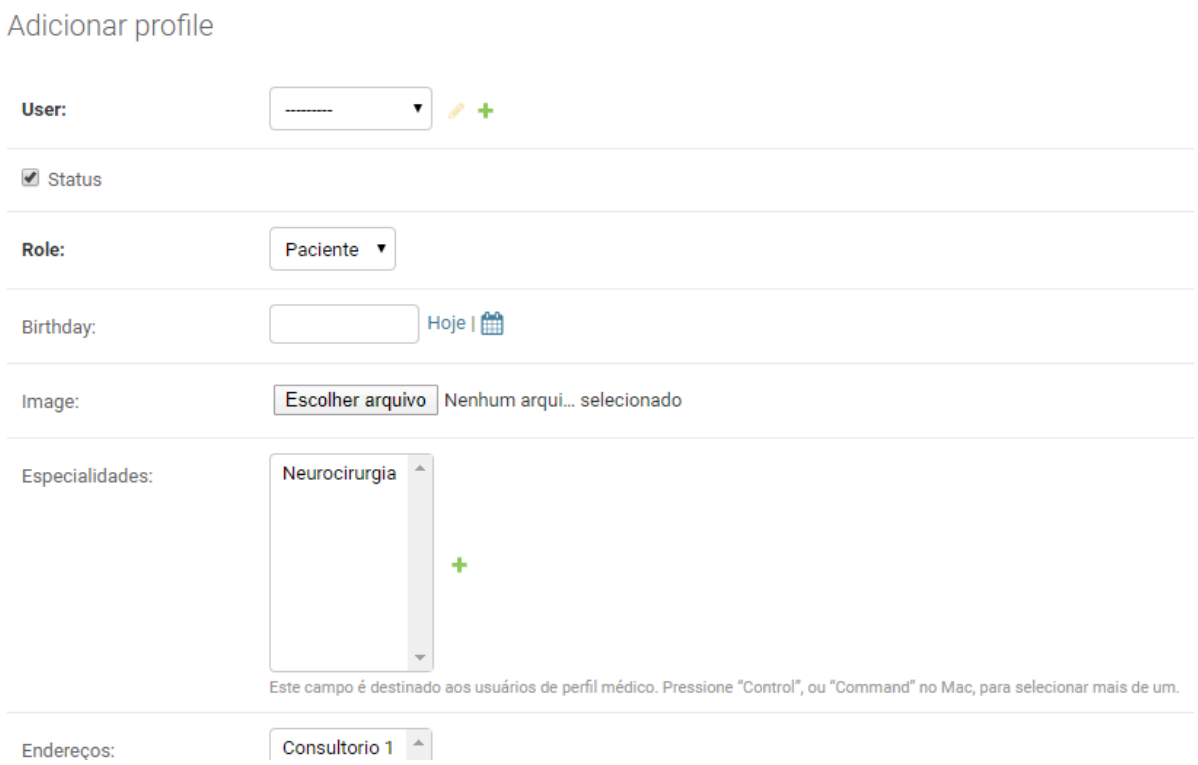


Figura 4.7: Customização do admin - Exclude

readonly_fields: deixa o campo apenas como leitura no formulário de edição e criação. Faremos isso aqui para que não seja permitido alterar o usuário atrelado a este perfil. Fique atento, pois ele aplica esse recurso na edição e na criação. Como a criação do perfil em nosso sistema é automática no momento em que criamos o usuário, o campo `user` pode ficar `readonly`, mas se precisássemos criar o perfil manualmente o campo `user` não poderia ficar como apenas leitura:

```
class ProfileAdmin(admin.ModelAdmin):
    readonly_fields = ('user',)

admin.site.register(Profile, ProfileAdmin)
```

Figura 4.8: Customização do admin - Readonly Fields

search_fields: lista de campos que poderão ser pesquisados na tela de listagem do admin. Nos campos que representam relacionamento precisam ser colocados o nome do campo com dois underlines e o nome do atributo que será pesquisado. Exemplo: `user__username` . Veja a seguir:

```
class ProfileAdmin(admin.ModelAdmin):
    search_fields = ('user__username',)

admin.site.register(Profile, ProfileAdmin)
```

Figura 4.9: Customização do admin - Search Fields

Para campos de tipo `integer` que possuem o atributo `choice`, por mais que os vejamos com nomes, como `role`, por exemplo, os valores salvos no banco de dados são inteiros, então a pesquisa não funcionará se buscarmos o valor `admin`; para funcionar precisaríamos buscar o valor 0. No exemplo anterior, buscamos um nome de usuário que por coincidência se chama `admin`, por isso funcionou, mas se quiséssemos buscar o tipo de usuário admin não seria a forma correta. Para isso, precisaríamos adicionar um `list_filter` de role assim: `list_filter = ('user__is_active', 'role')`.

4.2 Customização avançada

Algumas customizações exigem um pouco mais de configuração, algumas delas para agrupar inputs no formulário ou para retornar valores de relacionamentos `ManyToMany` na listagem do model admin. Vamos ver a seguir algumas customizações mais avançadas que podemos fazer no Django.

fieldsets: é similar ao `field`, porém aqui podemos agrupar os campos no formulário para que ele fique dividido de forma

organizada na tela:

```
class ProfileAdmin(admin.ModelAdmin):
    fieldsets = (
        ('Usuário', {
            'fields': ('user', 'birthday', 'image')
        }),
        ('Função', {
            'fields': ('role',)
        }),
        ('Extras', {
            'fields': ('specialties', 'addresses')
        }),
    )

admin.site.register(Profile, ProfileAdmin)
```

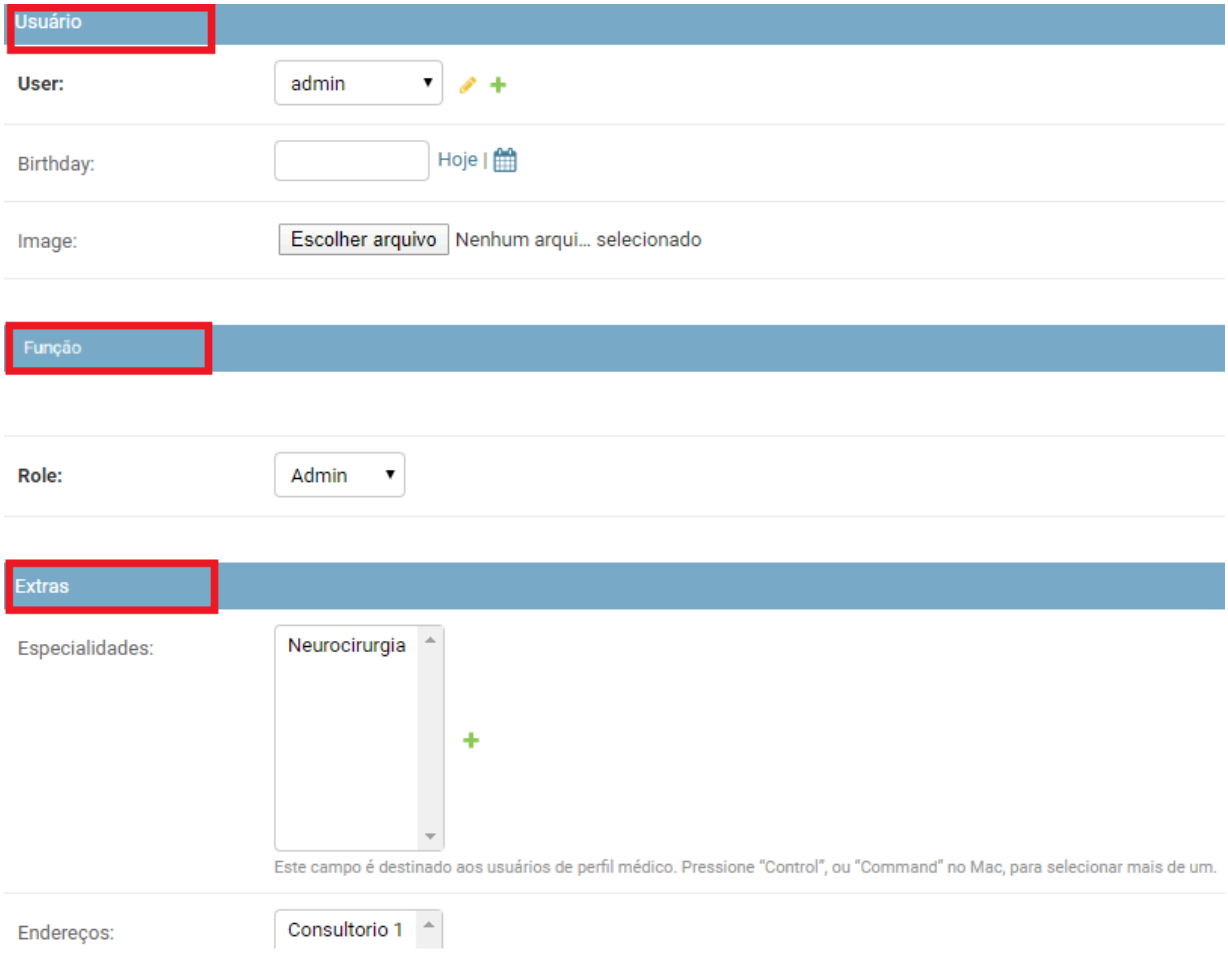


Figura 4.10: Customização do admin - Fieldsets

custom *list_display*: podemos customizar a exibição de um campo no `list_display` simplesmente criando um método para ele. Desse modo, podemos tratar o valor que desejamos que seja exibido na listagem do model admin:

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'birth',)

    def birth(self, obj):
        if obj.birthday:
            return obj.birthday.strftime("%d/%m/%Y")

admin.site.register(Profile, ProfileAdmin)
```

O método que criamos passa a poder ser adicionado ao `list_display`. Repare que passamos como parâmetro o atributo `self`, que no caso é o próprio escopo do objeto da classe `ProfileAdmin` e, depois, passamos o parâmetro que representará a instância do objeto da model `Profile`. Podemos chamá-lo como quisermos, mas geralmente o veremos como `obj` nas documentações do `django`. Através do `obj` poderemos acessar qualquer atributo da model em questão. Em nosso caso, nós acessamos o atributo `birthday` dizendo para o Django que no campo `birth` exibiremos o campo `birthday` da nossa model. Dentro desse método, podemos fazer o que quisermos, tendo ao final o comando `return` para retornar um valor que será listado na model admin.

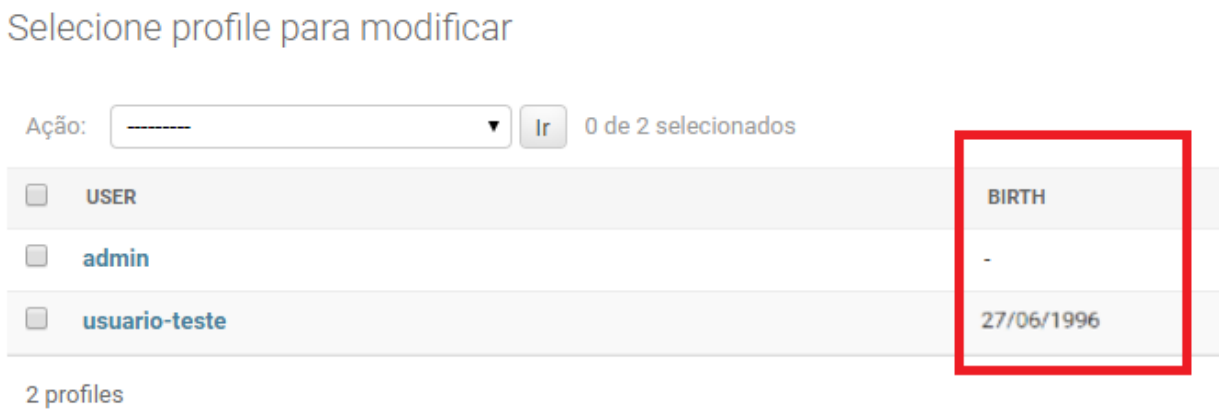


Figura 4.11: Customização do admin - Custom List Display

custom *empty_value_display*: também podemos pegar esse método que customiza o campo do `list_display` e personalizar a forma com que o exibiremos quando estiver vazio. Essa personalização do `empty_value_display` customizada precisa estar após o método de criação do campo customizado:

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'birth',)

    def birth(self, obj):
        return obj.birthday
    birth.empty_value_display = '___/___/_____'
```

```
admin.site.register(Profile, ProfileAdmin)
```

Figura 4.12: Customização do admin - Custom Empty Value Display

custom *list_display ManyToMany*: algo que ainda não fizemos foi listar os campos que são do tipo `ManyToMany` em nossa aplicação. Para fazermos isso, precisamos apenas criar os campos customizados da mesma maneira que no exemplo do `birth`, porém com pequenas diferenças. Vamos lá.

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'specialtiesList', 'addressesList',)

    def specialtiesList(self, obj):
        return [i.name for i in obj.specialties.all()]
    def addressesList(self, obj):
        return [i.name for i in obj.addresses.all()]

admin.site.register(Profile, ProfileAdmin)
```

O que fizemos aqui foi um básico de Python, onde iteramos todas as especialidades e endereços que temos atribuídos em nosso perfil. Essa iteração colocará esses dados em uma lista que será retornada pelos métodos `specialtiesList` e `addressesList`. Desse modo, podemos chamar esses dois métodos dentro da propriedade

`list_display` e assim nossas especialidades e endereços serão exibidos no painel admin na edição de perfil.

Perceba que os métodos são idênticos em sua estrutura, a única diferença é que o primeiro (`specialtiesList`) itera as especialidades e o segundo (`addressesList`), os endereços. Agora veja o resultado na imagem a seguir:

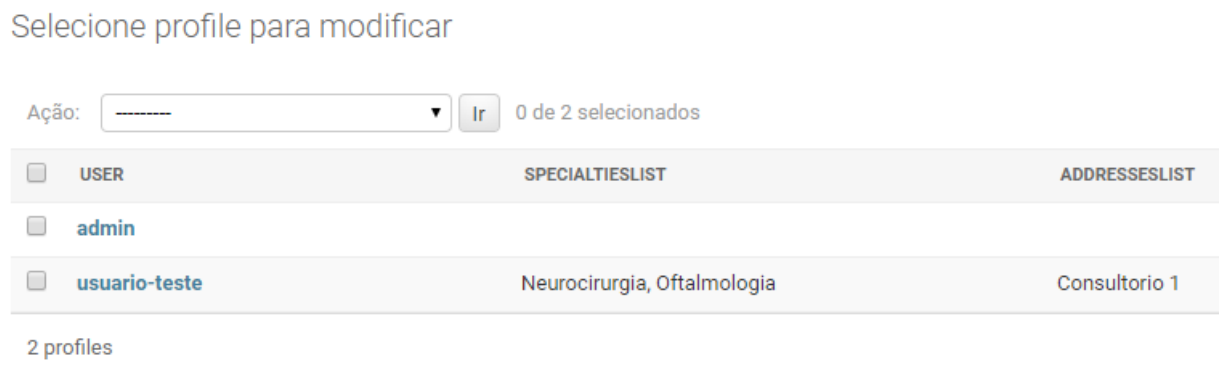


Figura 4.13: Customização do admin - Custom Empty Value Display

Adicionando arquivos: também podemos adicionar arquivos `css` e `js` em nosso admin. Os arquivos precisarão estar na pasta `static` do nosso sistema. Mais à frente aprenderemos a criar esse diretório dentro do projeto, por ora veremos apenas como adicionar o arquivo à model admin.

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'specialtiesList', 'addressesList',)

    class Media:
        css = {
            "all": ("css/custom.css",)
        }
        js = ("js/custom.js",)

admin.site.register(Profile, ProfileAdmin)
```

Em todos os exemplos foram mostrados apenas o método e sua aplicação, mas tenha em mente que isso precisa ser criado no arquivo `admin.py`. Caso seja criado em outro arquivo, você precisará importar o arquivo para o `admin.py` e registrar o método de customização que for criado.

Neste capítulo, aprendemos a criar uma model admin customizada para nosso painel administrativo. Isso nos permitirá deixar nosso sistema mais completo e mais intuitivo.

Django avançado

Figura 1: Django avançado

Esta unidade abordará de forma avançada recursos do Django que nos permitem ter uma aplicação mais robusta e profissional, deixando o sistema mais maduro e consistente através de ferramentas, como o ORM, customização dos formulários e das `views` com seus `templates` e também a criação de `middlewares` que nos permitirão monitorar todas as ações que ocorrem em nossa aplicação.

Capítulo 5
Trabalhando com Views e Urls

Neste capítulo, aprenderemos a criar nossas primeiras `views`. Como sabemos, as `views` são as responsáveis por fazer a conexão entre as `models` e os templates de uma aplicação, sendo que cada `view` possui um endereço de `url` ou `endpoint`. Este segundo nome é usado quando estamos criando uma `view` que deverá se comportar

como uma API Rest. Aqui criaremos as `views` que existirão em nossa aplicação.

As `views` possuem duas formas de serem acionadas. A primeira é através de uma requisição `HTTP` e a segunda, via `AJAX`, sendo que o navegador (`Client`) requisita a `view` que está no servidor (`Server`).

Independente da forma como é acionada, ela se comunicará com as models necessárias da aplicação e realizará a tarefa solicitada (não foque em entender como, por enquanto, estamos vendo o conceito). Após realizar o que for necessário, a `view` realiza seu retorno de resposta ao navegador (`Client`) com um resultado.

De acordo com a forma como ela é acionada (`HTTP` ou `AJAX`), teremos um formato de resposta diferente. No caso de uma requisição `HTTP`, a resposta poderá ser a renderização de uma página `HTML` e no caso de uma requisição `AJAX`, a resposta pode ser no formato de um `JSON` ou um `XML`.

Vamos ver a seguir uma imagem que reflete um exemplo de requisição feita por `HTTP` e uma por `AJAX`.

Exemplo com requisição HTTP:

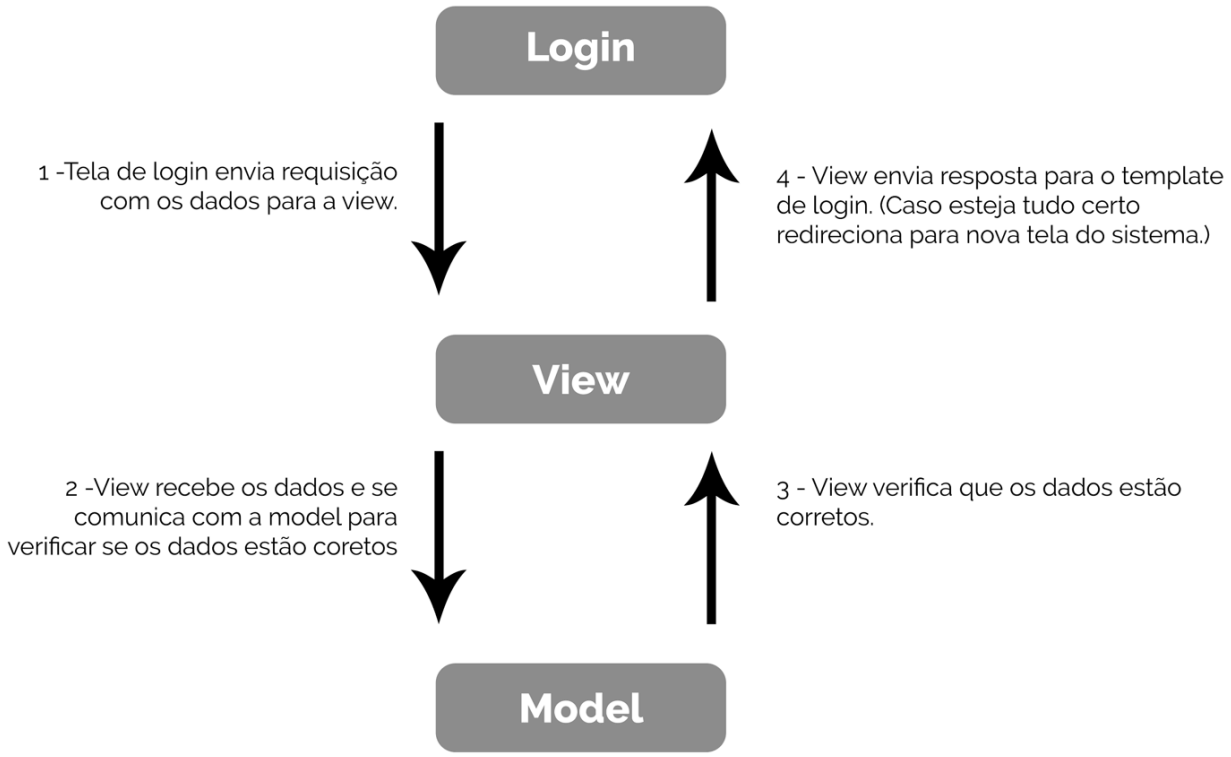


Figura 5.1: Conceito de view - View HTTP

Exemplo com requisição AJAX:

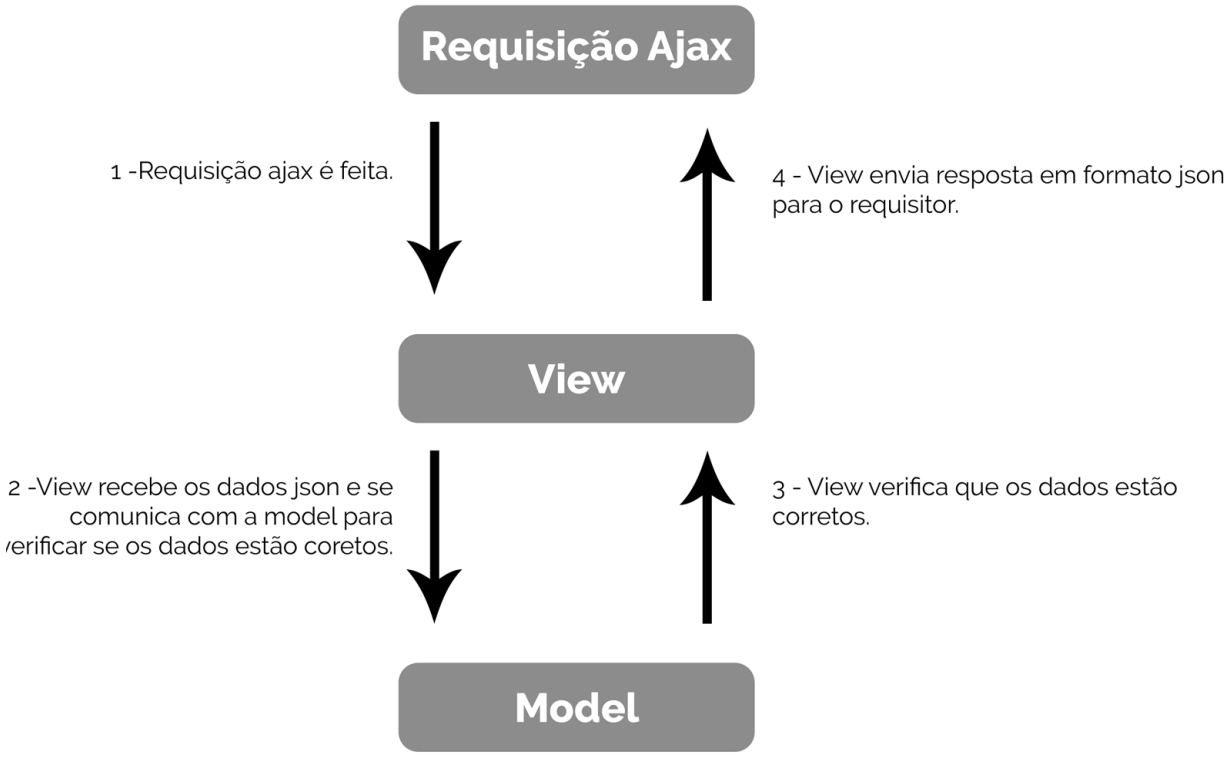


Figura 5.2: Conceito de view - View Ajax

Vamos ver também uma imagem que reflete bastante a estrutura de trabalho padrão do Django:

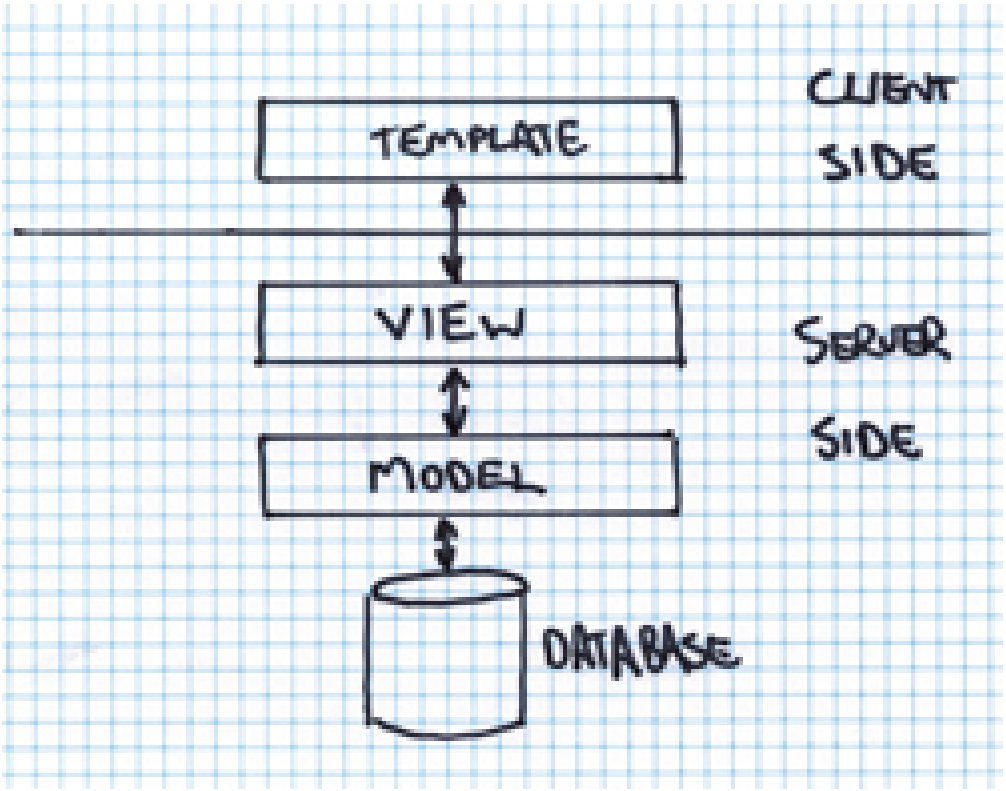


Figura 5.3: Estrutura do Django

Vamos colocar a mão na massa.

5.1 Criando a primeira view

Nesta etapa, criaremos uma `view` básica, apenas para entender como funciona o conceito de sua criação. Após isso, aplicaremos essa `view` nos capítulos seguintes e também criaremos outras, juntando o conhecimento adquirido.

Configurando nossos arquivos

A hierarquia atual do nosso projeto é esta a seguir:

```
livro_django
├──medicSearch
```

```
│        migrations
│        models
│        │     __init__.py
│        │     Address.py
│        │     DayWeek.py
│        │     Profile.py
│        │     Rating.py
│        │     Speciality.py
│        │
│        __init__.py
│        admin.py
│        apps.py
│        tests.py
│        views.py
├──medicSearchAdmin
│        media
│        settings
│        │       settings.py
│        │       development.py
│        │       production.py
│        │       testing.py
│        __init__.py
│        db.sqlite3
│        urls.py
│        wsgi.py
├──venv
├──manage.py
└──requirements.txt
```

Como podemos ver, dentro da pasta `medicSearch` temos um arquivo chamado `views.py`. É nele que se costuma colocar todo o código da view, porém não é uma boa prática fazer isso, pois, com o tempo, sua aplicação vai crescendo muito e um único arquivo pode ficar bem grande. Por mais que possamos criar nossos `apps` dentro do projeto para separar nosso código, essa separação deve ser feita apenas para organizar as necessidades de cada app e não para diminuir a quantidade de código que um ou mais arquivos possui.

Para organizar nosso código da forma correta, vamos remover o arquivo `views.py` e criar no lugar dele uma pasta chamada `views`. Dentro dela, vamos adicionar um arquivo `__init__.py`. Veja a seguir como esperamos que fique a nova estrutura:

```
livro_django
├──medicSearch
│        migrations
│        models
│        │        __init__.py
│        | ... Arquivos ocultos para otimizar a página
│        views
│        │        __init__.py
│        __init__.py
│        admin.py
│        apps.py
│        tests.py
├──medicSearchAdmin
│        media
│        settings
│        ... Arquivos ocultos para otimizar a página
```

Agora que nossa estrutura está pronta para receber nossas `views`, vamos criar nosso primeiro arquivo `view` e chamá-lo de `HomeView.py`. Nossa view principal terá como objetivo permitir que logo de primeira realizemos pesquisas de médicos com base em endereço e especialidade. Coloque esse arquivo dentro de `livro_django/medicSearch/views` e adicione o seguinte código a ele:

livro_django/medicSearch/views/HomeView.py

```
from django.http import HttpResponse

def home_view(request):
    return HttpResponse('<h1>Olá mundo!</h1>')
```

Vamos entender um pouco o que cada linha representa:

- **from django.http import HttpResponse**: aqui estamos importando a classe `HttpResponse` para dentro do nosso arquivo `HomeView.py`, desse modo poderemos usá-lo para retornar uma resposta para nosso `client`.
- **home_view()**: esse será o método que vamos disparar através da url `/` ou `/home`. Ainda não aprendemos isso, mas será nossa próxima etapa.
- **request**: parâmetro padrão da `view`. Esse parâmetro recupera dados da requisição, como os dados do usuário logado, caso haja alguém logado no sistema, ou dados da requisição, caso tenha sido um `GET` ou um `POST`. Os valores enviados nessa `GET` ou `POST`.

Configurando o `__init__.py` da view

Para que essa view funcione corretamente em nossa aplicação precisamos de mais algumas configurações. Uma delas é adicionar o arquivo `HomeView.py` dentro do `__init__.py`. Vamos alterar nosso arquivo `__init__.py`. Acesse-o dentro da pasta `livro_django/medicSearch/views` e realize as seguintes alterações:

livro_django/medicSearch/views/__init__.py

```
from .HomeView import *
```

Aqui estamos dizendo ao Django que, entre todos os arquivos que estão na pasta `views`, apenas os que estiverem importados dentro de `__init__.py` poderão ser usados dentro da aplicação, ou seja, se dentro da pasta `views` tivermos outros arquivos, nesse momento, apenas `HomeView.py` poderá ser usado pela aplicação como uma view direta. Essa é uma forma de segurança que o Python possui e da qual o Django utiliza para um melhor desempenho da aplicação, chamando apenas os arquivos necessários para serem usados.

Agora precisamos configurar nossas urls e para isso precisamos fazer algumas modificações em nosso código.

Configurando a pasta de urls

Se você voltar algumas páginas, onde mostramos a estrutura completa do projeto, vai perceber que dentro da pasta `medicSearchAdmin` temos um arquivo chamado `urls.py`, dentro do qual se encontram todas as urls do projeto. Poderíamos simplesmente adicionar uma linha a mais nele para criar a url da home, mas quando pensamos em *Engenharia de software* vemos que, ao escalarmos esse programa, o arquivo de urls da pasta `medicSearchAdmin` poderia ficar grande, então faremos com as urls o mesmo que fizemos com as `views`. Vamos criar uma pasta chamada `urls` dentro da pasta `medicSearch` e, dentro dela, o arquivo `__init__.py` e também o primeiro arquivo de urls, chamado `HomeUrls.py`. Vamos ver como nossa estrutura nova ficará:

```
livro_django
├──medicSearch
|        migrations
|        models
|        urls
|        |        __init__.py
|        |        HomeUrls.py
|        views
|        ... Arquivos ocultos para otimizar a página
```

Não apague os outros arquivos que não estão sendo exibidos aqui, eu os ocultei apenas para não ocupar muito espaço.

Com os arquivos criados, precisamos configurá-los. Vamos primeiro alterar o arquivo `__init__.py`. Perceba que para cada pasta que estamos configurando estamos utilizando esse arquivo para dar as permissões de importação. Vamos abri-lo e alterá-lo para que o Django tenha permissão para acessar o arquivo `HomeUrls.py`.

livro_django/medicSearch/urls/__init__.py

```
from .HomeUrls import *
```

Agora que o Django pode acessar nosso arquivo `HomeUrls.py` vamos configurá-lo criando nossa primeira url. Abra o arquivo `HomeUrls.py` dentro da pasta `urls` :

livro_django/medicSearch/urls/HomeUrls.py

```
from django.urls import path
from medicSearch.views.HomeView import home_view

urlpatterns = [
    path("", home_view),
]
```

Vamos entender um pouco o que esse arquivo está fazendo:

- **from django.urls import path**: aqui estamos importando o método `path` , que é o responsável por criar a url que acionará a view `HomeView` . O primeiro parâmetro é a `string` que corresponde ao caminho da url. O segundo corresponde ao método `view` que estamos chamando, em nosso caso, o `home_view` , que está dentro da view `HomeView.py` . Veja que não passamos os parâmetros do método.

Fique atento! Não se deve passar a primeira barra `"/"` quando criamos a url principal, o correto deve ser assim `""` . Para as demais urls devemos passar como nesse exemplo: `"caminho1/caminho2/"` .

Veja também que se não passarmos a barra no final do caminho teremos o seguinte erro:

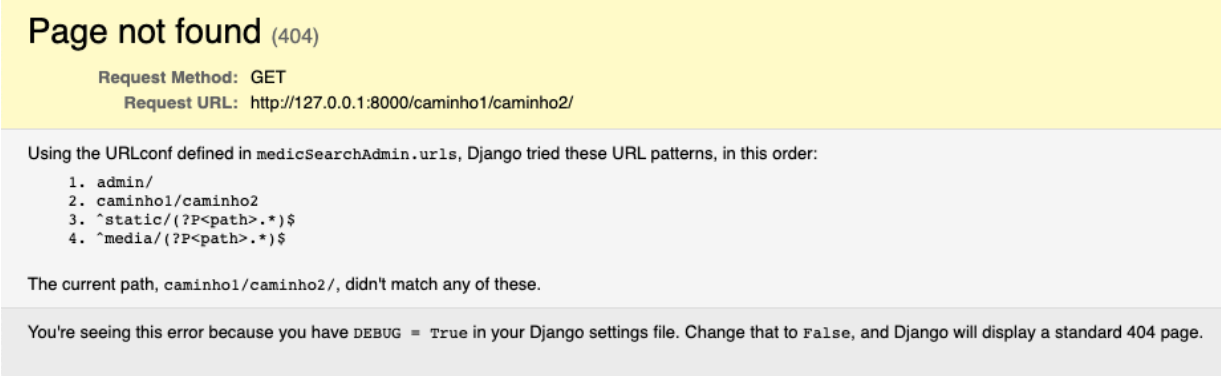


Figura 5.4: Barra final

Por isso, com exceção do caminho principal, precisamos sempre passar a barra no final do caminho, pois o Django, por padrão, sempre coloca uma barra no final da url requisitada. Assim, se você deixar sempre com barra no final, não terá problemas.

Agora vamos abrir o arquivo `urls.py` que fica dentro de `medicSearchAdmin` e alterar conforme o código a seguir:

livro_django/medicSearchAdmin/urls.py

```
from django.contrib import admin
from django.urls import path
from django.conf.urls.static import static
from django.conf import settings
# Adicione o import a seguir no código
from django.conf.urls import url, include

urlpatterns = [
    path('admin/', admin.site.urls),
    # Adicione a linha a seguir
    path('', include('medicSearch.urls.HomeUrls')),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT) +
static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Como podemos perceber, abaixo da linha `path('admin/', admin.site.urls)` nós adicionamos o path que incluirá nosso arquivo `HomeUrls.py`, que está dentro da pasta `medicSearch/urls`. Toda vez

que criarmos um novo arquivo de url, ele precisará ser incluído dentro desse arquivo de `urls.py` principal.

Se quisermos, podemos customizar o prefixo da url no arquivo principal de urls alterando para algo como no exemplo (não altere o arquivo como está a seguir, é apenas uma demonstração):

livro_django/medicSearchAdmin/urls.py

```
path('admin/', admin.site.urls),
path('principal/', include('medicSearch.urls.HomeUrls')),
```

Desse modo, todas as urls que estiverem sendo criadas dentro de `medicSearch/urls` terão como prefixo o caminho `principal/` . Mas vamos deixar como fizemos primeiro, sem prefixo nesse caso.

Testando a view

Podemos testar nossa primeira `view` e teremos o seguinte resultado na tela acessando o caminho http://127.0.0.1:8000/

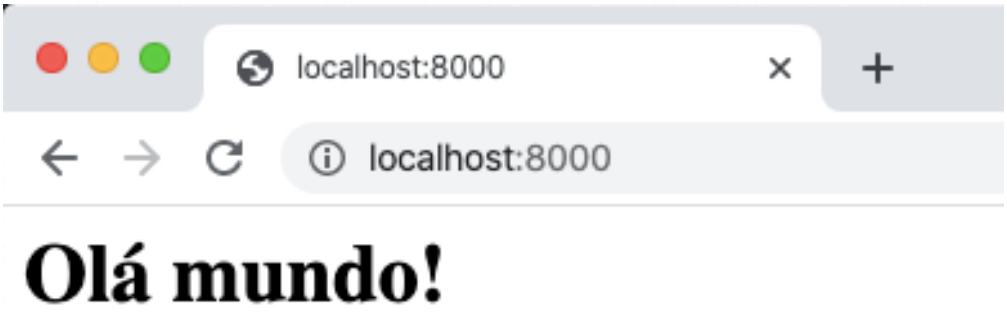


Figura 5.5: Primeira view

> Se por algum motivo a aplicação apresentar erro, pare o runserver e reexecute.

Caso queiramos, podemos passar o status do nosso retorno, por exemplo, adicionando o parâmetro `status` no método `HttpResponse`. Veja o exemplo a seguir:

```
from django.http import HttpResponse

def home_view(request):
    return HttpResponse('<h1>Olá mundo!</h1>', status=200)
```

Parabéns! Sua primeira `view` foi criada com sucesso. Nos capítulos adiante, vamos criar `views` utilizando todos os conceitos que aprendemos aqui e outros que ainda veremos.

5.2 Customizando urls no Django

Agora veremos como criar uma url customizada para nossa aplicação. Para isso, vamos criar uma `view` que futuramente será nossa `view` de perfil, ou seja, ela retornará como resultado nossos dados de perfil em uma tela `html`. Essa tela será criada capítulos à frente, mas vamos preparar a `view`.

Crie dentro da pasta `livro_django/medicSearch/views/` um arquivo chamado `ProfileView.py` e adicione o seguinte código nele:

livro_django/medicSearch/views/ProfileView.py

```
from django.http import HttpResponse

def list_profile_view(request, id=None):
    return HttpResponse('<h1>Usuário de id %s!</h1>' % id)
```

Por enquanto, vamos apenas exibir o id do usuário. Mais à frente esse id será necessário para a exibição do perfil do usuário, que no caso será de algum médico ou o seu próprio.

Não podemos esquecer de adicionar essa `view` como permitida dentro do arquivo `__init__.py`, que fica dentro da mesma pasta que as `views`. O código deverá ficar assim:

livro_django/medicSearch/views/__init__.py

```
from .HomeView import *
from .ProfileView import *
```

Agora vamos criar um arquivo de urls dentro da pasta `livro_django/medicSearch/urls` chamado `ProfileUrls.py` e adicionar o código a seguir:

livro_django/medicSearch/urls/ProfileUrls.py

```
from django.urls import path
from medicSearch.views.ProfileView import list_profile_view

urlpatterns = [
    path("", list_profile_view),
    path("<int:id>", list_profile_view),
]
```

Como podemos perceber, existem duas linhas de url chamando a mesma view `list_profile_view`. Mas isso é possível? Sem dúvidas, é possível e muito recomendável. O primeiro `path` chamará o método `list_profile_view` e não passará um `id`. Isso não é um problema, pois quando criamos esse método na `view` `ProfileView.py` nós falamos para o Python que `id` poderia ser `None`. Veja como escrevemos: `def list_profile_view(request, id=None):`, então caso chamemos a url sem o parâmetro `id`, ele será `None`, mas não se esqueça de que foi preciso colocar `id` como `None`.

O segundo `path` recebe uma url customizada, onde temos o recebimento de um valor inteiro que está nomeado como id,

exatamente aquele que criamos como parâmetro do método `list_profile_view`. Nesse caso, o `id` não será `None`, pois estamos passando um valor para ele.

Podemos passar quantos parâmetros quisermos na url. Poderia ser algo assim `<int:id>/<int:idade>/<int:ano>` e mais o que quisermos, mas se nossa url for assim, nosso método terá que ser algo como `def metodo_da_view(request, id, idade, ano)`. Vale lembrar que o primeiro parâmetro, que costumamos chamar de `request`, é o que trará os dados da requisição e os dados do usuário logado no sistema.

Agora que nossa url está configurada, precisamos importar o arquivo `ProfileUrls.py` dentro do arquivo `__init__.py` que fica na pasta `livro_django/medicSearch/urls`.

livro_django/medicSearch/urls/__init__.py

```
from .HomeUrls import *
from .ProfileUrls import *
```

Pronto! Nosso arquivo está preparado para ser incluído no arquivo de `urls` principal. Abra o arquivo `urls.py` que fica dentro da pasta `livro_django/medicSearchAdmin` e faça a seguinte alteração.

livro_django/medicSearchAdmin/urls.py

```
from django.contrib import admin
from django.urls import path
from django.conf.urls import url, include
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('medicSearch.urls.HomeUrls')),
    # Adicione a linha a seguir
    path('profile/', include('medicSearch.urls.ProfileUrls')),
```

```
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT) +
static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Perceba que adicionamos como prefixo `profile/`, desse modo todas as urls criadas dentro do arquivo `ProfileUrls.py` terão o prefixo `profile/`. Se rodarmos nossa aplicação agora, conseguiremos ver nossa nova view funcionando. Veja só.

1. `http://localhost:8000/profile/3`

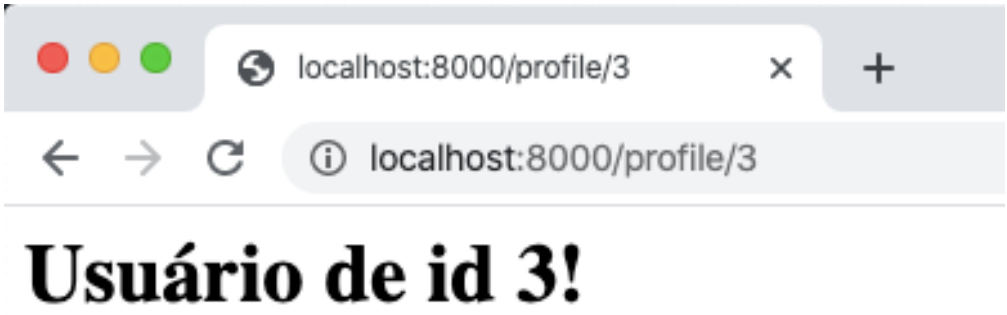


Figura 5.6: View de perfil

2. `http://localhost:8000/profile`

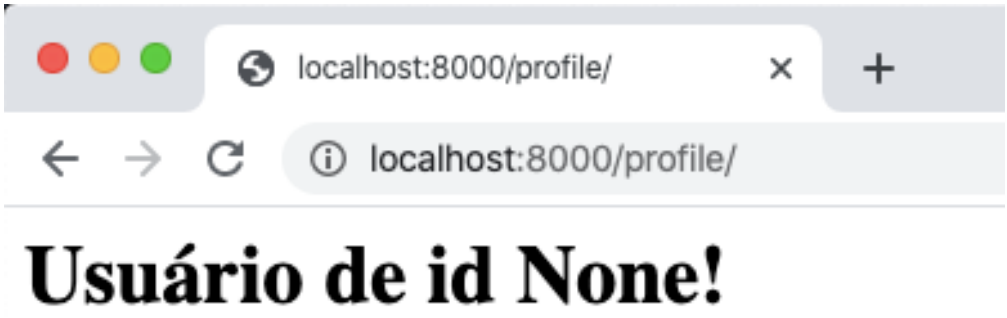


Figura 5.7: View de perfil None

Quando não passarmos o id, podemos fazer uma modificação para retornar o id do usuário que está logado. Para isso, devemos usar o parâmetro `request` que até agora não utilizamos, porém sabemos que ele contém todos os dados do usuário que está logado no sistema. Vamos fazer uma pequena alteração no código da `view` :

livro_django/medicSearch/views/ProfileView.py

```
from django.http import HttpResponse

def list_profile_view(request, id=None):
    if id is None and request.user.is_authenticated:
        id = request.user.id
    elif not request.user.is_authenticated:
        id = 0

    return HttpResponse('<h1>Usuário de id %s!</h1>' % id)
```

Estamos dizendo para a `view` que, caso o id seja `None` , vamos retornar como id o id do usuário que está logado. Para acessar

qualquer dado padrão do usuário logado é só acessar `request.user`. Alguns dados que podem ser do seu interesse:

- **request.user.username**: nome de usuário.
- **request.user.email**: e-mail do usuário.
- **request.user.first_name**: primeiro nome.
- **request.user.last_name**: último nome.
- **request.user.date_joined**: data de cadastro.
- **request.user.is_active**: se está ativo ou inativo.
- **request.user.is_staff**: se o usuário pode acessar o site do administrador.
- **request.user.is_superuser**: se é um superusuário com acesso ao painel admin do Django.
- **request.user.last_login**: data e hora do último login do usuário.
- **request.user.is_authenticated**: atributo que informa se um usuário está logado ou não.

Todos esses dados são preenchidos na edição de usuário como vimos nos capítulos anteriores.

5.3 Nomes dinâmicos para os links

Algo que pode ser interessante de usarmos no futuro são os nomes dinâmicos no link. Eles são muito utilizados para que você tenha facilidade em alterar um link sem precisar trocá-lo na aplicação inteira. Você dá um nome ao link como uma variável e utiliza esse nome nos arquivos de html. Vamos ver a seguir como criá-los.

Abra o arquivo `ProfileUrls.py` que fica na pasta `livro_django/medicSearch/urls` e vamos adicionar os nomes às urls:

livro_django/medicSearch/urls/ProfileUrls.py

```
from django.urls import path
from medicSearch.views.ProfileView import list_profile_view

# Adicione o atributo name e o valor nas linhas a seguir:
urlpatterns = [
    path("", list_profile_view, name='profiles'),
    path("<int:id>", list_profile_view, name='profile'),
]
```

O atributo `name` permite que criemos um nome para a url, desse modo, caso precisemos trocar a url da view, não será necessário mexer em cada template da aplicação, pois esse `name` será como uma variável.

Alguns capítulos à frente você verá que aplicaremos esses nomes aos templates da aplicação.

Existem diversas outras maneiras de trabalhar com urls no Django, mas esta sem dúvidas é a mais usada e recomendada. Adiante, utilizaremos bastante do que aprendemos neste capítulo para estruturar nossa aplicação.

Capítulo 6
Django ORM

Neste capítulo, aprenderemos a trabalhar com a ferramenta ORM do Django.

Mapeamento objeto-relacional (*Object-relational mapping* - ORM) é uma técnica para aproximar o paradigma de desenvolvimento de aplicações orientadas a objetos do paradigma do banco de dados relacional. Tem sido muito utilizada atualmente e pode trazer muitas vantagens para quem a implementa, tais como: fácil reutilização do código, manutenibilidade e legibilidade da estrutura da aplicação. Através do uso da Orientação a Objetos, conseguiremos criar e mapear um objeto para que ele se comporte da forma adequada em uma estrutura de banco de dados relacional.

No capítulo 3, já aprendemos a trabalhar com `models`, as estruturas que refletem o objeto da aplicação de forma relacional na estrutura do nosso banco de dados. Isso só ocorre graças às técnicas de ORM que o Django implementa por padrão em sua estrutura de aplicação.

Agora criaremos nossas `views` e nelas implementaremos uma série de métodos que o Django nos proporciona para trabalhar com as técnicas ORM. Vamos lá.

6.1 Consultas no Django com QuerySet

Nesta etapa, criaremos todas as `views` de pesquisa que nossa plataforma terá. Vamos começar com a mais simples de todas: a pesquisa por um médico.

Assim como foi feito no capítulo sobre `views`, precisaremos criar uma `view` para o médico, um arquivo de url, e incluí-lo dentro do

arquivo de urls principal. Vamos lá. Crie um arquivo chamado `MedicView.py` dentro da pasta `livro_django/medicSearch/views` e adicione o código a seguir:

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse

def list_medics_view(request):
    return HttpResponse('Listagem de 1 ou mais médicos')
```

Por enquanto, estamos apenas criando a estrutura para que ela exista, mais à frente vamos adicionar consultas do Django QuerySet nessa estrutura para que ela consiga nos entregar o resultado que esperamos, que é uma busca de médicos por especialidade, nome e/ou localidade. Importe esse arquivo que acabamos de criar dentro do `__init__.py` que fica dentro da pasta de `views`.

livro_django/medicSearch/views/__init__.py

```
from .HomeView import *
from .ProfileView import *
from .MedicView import *
```

Agora precisamos criar um arquivo de urls para o médico. Dentro da pasta `livro_django/medicSearch/urls`, crie um arquivo chamado `MedicUrls.py` e adicione o código a seguir:

livro_django/medicSearch/urls/MedicUrls.py

```
from django.urls import path
from medicSearch.views.MedicView import list_medics_view

urlpatterns = [
    path("", list_medics_view, name='medics'),
]
```

Também precisamos importar nosso arquivo `MeidicUrls.py` dentro do arquivo `__init__.py` que fica na pasta `livro_django/medicSearch/urls`. Vamos lá:

livro_django/medicSearch/urls/__init__.py

```
from .HomeUrls import *
from .ProfileUrls import *
from .MedicUrls import *
```

Com isso feito, vamos alterar o arquivo de urls principal e incluir o arquivo `MedicUrls` em nosso `path` de urls.

livro_django/medicSearchAdmin/urls.py

```
from django.contrib import admin
from django.urls import path
from django.conf.urls import url, include
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('medicSearch.urls.HomeUrls')),
    path('profile/', include('medicSearch.urls.ProfileUrls')),
    path('medic/', include('medicSearch.urls.MedicUrls')),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT) +
static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

> Esse processo de criar arquivos da `view` e das urls parece ser repetitivo, mas na realidade estamos mantendo o padrão de nomenclatura no projeto, desse modo fica mais fácil realizar manutenção no código.

Pronto, agora temos nossa `view` de médicos criada e estamos prontos para criar as buscas. Abra o arquivo `MedicView.py` e vamos começar a trabalhar nele:

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse

def list_medics_view(request):
```

```
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    return HttpResponse('Listagem de 1 ou mais médicos')
```

O primeiro passo é capturar o que virá na url. Em nosso caso, para as consultas, usaremos o método `GET`, pois assim nossas buscas poderão ser compartilhadas via `url`. Sendo assim, teremos quatro parâmetros que poderão ser passados em nossa url, mas apenas um precisa ser obrigatório, ou seja, se já houver um desses quatro parâmetros, os outros três já não serão necessários; porém, quanto mais parâmetros houver, mais precisa será a busca. Vamos ver na lista a seguir os possíveis parâmetros que podemos passar. Vou listá-los separados, mas podemos juntá-los em uma única requisição caso desejemos.

- http://localhost:8000/medic/?neighborhood=1
- http://localhost:8000/medic/?city=1
- http://localhost:8000/medic/?state=1
- http://localhost:8000/medic/?speciality=1
- http://localhost:8000/medic/?name=marcus+vinicius

Exemplo de url com mais de um parâmetro:

- http://localhost:8000/medic/?name=marcus+vinicius&speciality=1&neighborhood=1&city=1&state=1

Para facilitar nosso trabalho, o bairro, a cidade e o estado serão passados como id, até para evitar problemas com locais que possam possuir o mesmo nome.

Agora vamos listar nossos dados usando o `ORM` do Django. Vamos alterar nosso arquivo `MedicView.py`:

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse
# Adicione a linha a seguir
from medicSearch.models import Profile

def list_medics_view(request):
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    # Adicione as linhas a seguir
    medics = Profile.objects.all()
    print(medics)

    return HttpResponse('Listagem de 1 ou mais médicos')
```

Se pegarmos esse código e tentarmos acessar qualquer uma daquelas urls listadas no passo anterior, veremos no console do Python um resultado parecido com este:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py runserver
127.0.0.1:8080
<QuerySet [<Profile: admin>, <Profile: usuario-teste>]>
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Para trazermos esses resultados foi necessário apenas importar a model `User` dentro do arquivo e chamá-la dessa maneira: `Profile.objects.all()`. Assim estamos pedindo ao Django que traga todos os dados que temos na tabela `Profile` da nossa aplicação. Sempre teremos uma estrutura parecida com esta: `nome_model.objects.quantidade_resultado()`.

Vejamos um outro exemplo em que a quantidade resultado não seria `all()`: `Profile.objects.first()`. Desse modo, só iríamos trazer o primeiro resultado da tabela.

6.2 Filtrando consultas no ORM do Django

Na maioria dos casos, precisaremos adicionar filtros em nossa consulta, para isso precisamos adicionar antes do método `first` ou `all` um método chamado `filter()` e como parâmetro chamamos o nome do atributo que desejamos utilizar para fazer o filtro, sabendo que cada atributo corresponde ao nome de uma coluna no banco de dados.

Se quisermos filtrar todos os perfis do tipo médico, por exemplo, passaremos `filter(role=2)`. Veja a seguir:

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse
from medicSearch.models import Profile

def list_medics_view(request):
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    # Altere a linha a seguir para este código
    medics = Profile.objects.filter(role=2).all()
    print(medics)

    return HttpResponse('Listagem de 1 ou mais médicos')
```

Como podemos ver, nessa `view`, estamos pedindo que o Django traga todos os perfis que tenham a `role` igual a 2, ou seja, todos os médicos, já que no `__init__.py` que fica dentro da pasta `models` temos a configuração do `ROLE_CHOICE` assim. Fizemos isso no capítulo que fala sobre models, vou deixar a seguir para relembrar:

livro_django/medicSearch/models/__init__.py

```
ROLE_CHOICE = (
    (1, 'Admin'),
    (2, 'Médico'),
    (3, 'Paciente')
)
```

Até aqui não temos muita coisa de especial em usar o ORM do Django, mas se pararmos para analisar, para fazermos uma consulta que busque o perfil dos médicos por um nome e um bairro, precisaríamos fazer um `join` entre as tabelas de `Profile`, `User` (que é a tabela padrão do Django), `Address` e `Speciality`, já que um perfil tem um usuário atrelado a ele e, quando o perfil é do tipo médico, podemos ter mais de um endereço e mais de uma especialidade atrelada a ele. Imagina o tamanho da consulta que teríamos que criar. Veja a seguir como podemos fazer isso utilizando o ORM, poupando-nos um imenso trabalho. Vamos começar criando o filtro do nome do médico:

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse
from medicSearch.models import Profile

def list_medics_view(request):
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    # Altere daqui em diante
    medics = Profile.objects.filter(role=2)

    if name is not None and name != '':
        medics = medics.filter(user__first_name=name)
    print(medics.all())

    return HttpResponse('Listagem de 1 ou mais médicos')
```

Como vimos, agora estamos verificando se `name` é nulo, caso não seja, usamos o `filter`. Mas observe que o parâmetro foi passado de uma maneira diferente. Isso acontece porque o campo `user` que criamos na model `Profile` é uma `foreing key`. Então precisamos passar o nome do atributo que criamos na model `Profile`, que no caso é `user`; após ele, precisamos adicionar dois underlines `__`; e depois, o nome do campo que desejamos usar como cláusula `where` do nosso filtro, parecido com o que fizemos na configuração do admin alguns capítulos atrás. Ou seja, `user__first_name` está pesquisando na tabela `User` o campo `first_name` e vendo quais profiles têm um `user` que possui o `first_name` que passamos como parâmetro.

Assim se filter for `filter(user__username="marcus")`, será verificado todo `Profile` que tem um `user` com `first_name` igual a `marcus`. Não se esqueça de que essa pesquisa não ignora maiúsculas e minúsculas, desse modo ele verificará exatamente o que foi escrito. Se quisermos usar algo como o `like` do SQL, precisamos alterar nossa consulta `medics.filter(user__first_name=name)`, adicionando o `__contains` ao término dela.

Agora, vamos fazer nossa busca filtrar por especialidade e bairro:

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse
from medicSearch.models import Profile

def list_medics_view(request):
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    medics = Profile.objects.filter(role=2)
    if name is not None and name != '':
        medics = medics.filter(user__first_name__contains=name)
```

```
    if speciality is not None:
        medics = medics.filter(specialties__id=speciality)
    if neighborhood is not None:
        medics = medics.filter(addresses__neighborhood=neighborhood)

    print(medics.all())
    return HttpResponse('Listagem de 1 ou mais médicos')
```

Desse modo, podemos filtrar nossos médicos pedindo que tenham a especialidade e bairro que desejamos. Vale lembrar que o bairro é um campo que está dentro da model `Address`, por isso foi possível realizar a consulta direta dele pelo atributo `addresses` que existe na model `Profile`. Para a cidade e o estado, precisaremos ter uma abordagem um pouco mais profunda, já que cidade está dentro de bairro e estado está dentro de cidade. Vamos ver a seguir:

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse
from medicSearch.models import Profile

def list_medics_view(request):
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    medics = Profile.objects.filter(role=2)
    if name is not None and name != '':
        medics = medics.filter(user__first_name__contains=name)
    if speciality is not None:
        medics = medics.filter(specialties__id=speciality)

    # Pule uma linha entre speciality e neighborhood para melhorar a
legibilidade e complemente o próximo if com o else que aparece a seguir
    if neighborhood is not None:
        medics = medics.filter(addresses__neighborhood=neighborhood)
    else: # Adicione daqui em diante
```

```
        if city is not None:
            medics = medics.filter(addresses__neighborhood__city=city)
        elif state is not None:
            medics =
medics.filter(addresses__neighborhood__city__state=state)

    print(medics.all())
    return HttpResponse('Listagem de 1 ou mais médicos')
```

Podemos ver como é simples realizar consultas com o ORM do Django. Apenas usando a model `Profile` foi possível chegarmos até as demais models que nem sequer possuem relacionamento direto com a `Profile`, como `City`, `State` e `Neighborhood`, mas que possuem relacionamento com alguma model que se relaciona com a `Profile`. Desse modo simples podemos fazer diversas outras consultas em nossa aplicação. Com o que fizemos já é possível obter resultados positivos, lembrando que é legal popular um pouco o sistema para que você consiga ver uma consulta efetiva ocorrendo.

> Você pode filtrar usando vírgulas ou usando várias vezes o método `filter`, como: `filter(param1=1, param2=2)` ou `filter(param1=1).filter(param2=2)`.

Vou deixar a seguir uma lista de possíveis filtros que podem ser aplicados no ORM do Django, lembrando que esses elementos sempre possuem o nome do atributo como predecessor:

- `__in` : Cláusula, IN(1,2,3). Exemplo: `.filter(id__in=[1,2,3])`;
- `__lt` : Cláusula menor que, <. Exemplo: `.filter(user__date_joined__lt=datetime.now())`;
- `__lte` : Cláusula menor ou igual, <=. Exemplo: `.filter(user__date_joined__lte=datetime.now())`;
- `__gt` : Cláusula maior que, >. Exemplo: `.filter(user__date_joined__lt=datetime.now())`;
- `__gte` : Cláusula maior ou igual, >=. Exemplo: `.filter(user__date_joined__lte=datetime.now())`;

- `__icontains` : Like. Exemplo: `.filter(user__first_name__icontains="Marcus")` ;
- `__startswith` : Inicia com. Exemplo: `.filter(user__first_name__startswith="Marcus")` ;
- `__startswith` : Filtra o ano daquela data. Exemplo: `.filter(user__date_joined__year='2016')` .

Também existem mais alguns métodos para filtrar que quero deixar para você:

- `all` : Retorna todas as linhas da consulta como um array de objetos da model consultada. Exemplo: `.all()` ;
- `.all()[:5]` : Limite do sql. Exemplo: `.all()[:5]` ;
- `first` : Retorna a primeira linha da consulta como um objeto da model consultada. Exemplo: `.first()` ;
- `filter` : Cláusula que filtra por uma condição. Exemplo: `.filter(id__in=[1,2,3])` ;
- `exclude` : Cláusula que exclui toda condição verdadeira. Exemplo: `.exclude(id=1)` ;
- `order_by` : Orderna por asc. Exemplo: `.order_by('user__date_joined')` ;
- `order_by` : Orderna por desc. Exemplo: `.order_by('-user__date_joined')` ;
- `Q()` : OR. Exemplo: `.filter(Q(user__first_name__startswith='R') | Q(user__last_name__startswith='D'))` .

Para usar o `Q()` , você precisa importar o `django.models` assim: `from django.db.models import Q` . Não esqueça que isso deve ser feito na parte superior do arquivo. Veja o código a seguir, onde colocamos o filtro para procurar o nome buscado tanto em `first_name` quanto em `username` :

livro_django/medicSearch/views/MedicView.py

```
from django.http import HttpResponse
from medicSearch.models import Profile
```

```
# Adicione o import a seguir no código
from django.db.models import Q

def list_medics_view(request):
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    medics = Profile.objects.filter(role=2)
    if name is not None and name != '':
        # Altere a linha a seguir
        medics = medics.filter(Q(user__first_name__contains=name) |
Q(user__username__contains=name))
    if speciality is not None:
        medics = medics.filter(specialties__id=speciality)

    if neighborhood is not None:
        medics = medics.filter(addresses__neighborhood=neighborhood)
    else:
        if city is not None:
            medics = medics.filter(addresses__neighborhood__city=city)
        elif state is not None:
            medics =
medics.filter(addresses__neighborhood__city__state=state)

    print(medics.all())
    return HttpResponse('Listagem de 1 ou mais médicos')
```

Com isso podemos ter uma busca mais abrangente.

Geralmente, o order_by fica após o filter e antes do first/all. Algo assim: `filter().order_by().all()` .

O exclude faz o oposto do filter, se criamos um filtro com `.exclude(id=1)` , ele vai excluir o id 1 do resultado da query.

Com todos esses itens você conseguirá criar consultas muito completas no ORM do Django.

Se quiser fazer testes é só adicionar os gets na url. Utilize valores que correspondem aos dados que tem no seu banco, por exemplo, se você só tem um `state` então provavelmente o id dele será 1, mas você pode ir até o admin e ver o id de `state`, `city` etc. para poder construir a consulta correta. Se preferir esperar, mais à frente vamos criar uma interface para nossa `view` que trará o resultado em um `html`.

6.3 Alterando dados com Django QuerySet

Para alterar dados com o QuerySet do Django, basta realizar a consulta como vimos anteriormente e escolher o que desejamos fazer com o resultado. Vamos ver a seguir alguns exemplos que não foram implementados ainda:

Create

```
from medicSearch.models.Speciality import Speciality
try:
    speciality = Speciality()
    speciality.name = "Endocrinologia"
    speciality.save()
except Exception as e:
    print("Um erro ocorreu ao salvar uma nova especialidade. Descrição %s"
% e)
```

Update

```
from medicSearch.models.Profile import Profile
try:
    medic = Profile.objects.filter(id=1).first()
    medic.user.first_name = "João"
    medic.user.last_name = "Victor"
```

```
    medic.user.save()
except Exception as e:
    print("Um erro ocorreu ao editar um usuário. Descrição %s" % e)
```

Delete

```
# Deletando uma especialidade
from medicSearch.models.Speciality import Speciality
try:
    Speciality.objects.filter(id=6).delete()
except Exception as e:
    print("Um erro ocorreu ao deletar uma especialidade. Descrição %s" %
e)

# Deletando um usuário através da model Profile
from medicSearch.models.Profile import Profile
try:
    profile = Profile.objects.filter(user__id=3).first()
    profile.user.delete()
except Exception as e:
    print("Um erro ocorreu ao deletar um usuário. Descrição %s" % e)
```

Transactions

Além desses métodos convencionais, podemos também criar queries com `transactions`. O `transaction` permite que haja mais segurança no caso de precisarmos salvar, alterar ou deletar mais de um elemento ao mesmo tempo. Caso alguma das ações falhe, todas as demais serão desfeitas, desse modo você previne erros como o de modificar em uma tabela e não na outra, evitando instabilidade na plataforma. Veja a seguir um breve exemplo:

Update

```
from medicSearch.models.Profile import Profile
from django.db import transaction, IntegrityError

try:
    with transaction.atomic():
        profile = Profile.objects.filter(id=1).first()
```

```
        profile.role = 3
        profile.user.first_name = "João"
        profile.user.last_name = "Victor"

        # Atualizando a tabela de Profile e User juntas
        profile.save()
        profile.user.save()
        # Não se esqueça de que profile.user está acionando o objeto
`User` sem precisar fazer uma query direta.
except IntegrityError as e:
    print("Erro ao editar as tabelas Profile e User. Descrição: %s" % e)
```

O módulo `transaction` possui o método `atomic()` que reverterá as atualizações com um `Rollback` caso haja alguma falha no `save` do `Profile` ou no `save` do `User`. O `Rollback` é um método conhecido em linguagem de banco de dados, pois é responsável por realizar a reversão do banco resetando as ações que funcionaram para voltarem ao estado inicial da ação, já que alguma das alterações não funcionou. Assim você poderá novamente tentar fazer a alteração.

> Existem várias situações que podem gerar falha em uma sequência de queries, por exemplo a queda de conexão do banco. O `transaction` será útil para evitar que algo ocorra somente pela metade.

Esses métodos serão muito usados mais à frente no livro, mas não tem mistério, o Django preza pela simplicidade e limpeza em seu código. Podemos perceber que foi possível até remover um usuário através da model `Profile` sem que precisássemos chamar a própria model `User`, que é uma model default do Django.

Com isso, aprendemos tudo que é necessário para trabalhar com a ferramenta de ORM do Django, uma das mais utilizadas pelos programadores Python.

Nos próximos capítulos, usaremos o que foi visto aqui para integrar nossos formulários de edição e pesquisa com nossa model, de forma

que consigamos alterar dados do nosso próprio perfil, adicionar médicos em nossa listagem de favoritos e realizar buscas em nosso html.

CAPÍTULO 7
Trabalhando com templates - Parte I

Por ser um pouco mais extenso, este assunto será dividido entre os capítulos 7, o atual, e o 8.

Agora começaremos a criação dos nossos templates html e para isso utilizaremos alguns recursos de `views` e do Django QuerySet.

Para nosso sistema teremos as seguintes telas `html` :

- **home.html**: onde serão feitas as consultas dos médicos.
- **medics.html**: lista com todos os resultados de médicos buscados.
- **profile.html**: onde podemos ver nosso perfil e o perfil de outros usuários do sistema.
- **edit-profile.html**: tela de edição do perfil.
- **login.html**: tela para fazer login no sistema.
- **register.html**: tela para fazer registro no sistema.
- **change-password.html**: tela onde será feita a alteração de senha.

Muitas dessas telas serão criadas neste capítulo, porém só implementaremos suas funcionalidades um pouco mais à frente.

7.1 Criando o nosso template base

Nessa primeira etapa, criaremos nosso arquivo `base` . Ele terá esse nome pois será o pai de todos os templates e é nele que vamos adicionar nossos arquivos `css` e `js` globais, nosso cabeçalho e nosso rodapé. Desse modo, não será necessário ficar criando cabeçalho e rodapé em todos os arquivos, até porque isso seria ruim

para a manutenção do sistema. Essa técnica é chamada de *Herança de template*.

Vamos criar uma pasta chamada `templates` dentro do diretório `livro_django/medicSearch` e, dentro dela, vamos criar o arquivo `base.html` e adicionar o código a seguir:

medicSearch/templates/base.html

```
{% load static %}
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
    {% block styles %}{% endblock %}
    <title>Busca Doutor - {% block title %}{% endblock %}</title>
  </head>
  <body>
    <header></header>
    {% block content %}{% endblock %}
    <footer></footer>
    {% block scripts %}{% endblock %}
  </body>
</html>
```

Vamos entender as linhas que adicionamos ao `html` que fazem parte do `django template`.

- **{% load static %}**: nessa linha estamos dizendo que o template será capaz de ler qualquer arquivo que esteja dentro da pasta `static`. Essa pasta ainda não foi criada, mas em breve veremos como fazer isso.
- **{% block %}{% endblock %}**: essa tag permite que criemos blocos dentro de um `template`, que servem para que possamos adicionar conteúdo ao `template` que está sendo estendido através daquele que o estende. Para ser mais claro, se o `template base.html` possui um `block title`, eu poderei

alterar seu `block title` de dentro do `template home.html`, que estenderá o `base.html`. Como vemos nesse arquivo `html`, foram criados quatro `blocks`: o primeiro, para que alteremos o título do html de dentro do `template` que estender `base.html`; o segundo, para que coloquemos arquivos `css` em nosso `template`; o terceiro, para conter o conteúdo `html` que desejamos que seja visto quando o navegador renderizar o conteúdo da página; e o quarto, para que adicionemos arquivos `js` em nosso `template`.

Na herança de `template`, o `template` filho estende o `template` pai dentro dele e assim todo conteúdo e todos os blocos que existem dentro do pai podem ser reutilizados no filho. Por isso, criar blocos dentro do pai é fundamental para termos uma organização correta do html.

Agora que entendemos como funciona o uso de `blocks`, vamos adicionar alguns arquivos `css` e `js` em nosso `template` base e também vamos completar nossas tags `header` e `footer`. Altere o arquivo `base.html` para ficar como o código a seguir:

medicSearch/templates/base.html

```
{% load static %}
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <!-- Adicione as linhas a seguir -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css">
    <link rel="stylesheet" href="{% static 'css/global.css' %}">
    <!-- Até aqui -->
```

```
    {% block styles %}{% endblock %}
    <title>Busca Doutor - {% block title %}{% endblock %}</title>
  </head>
  <body>
    <header>
        <!-- Adicione o código a seguir dentro da header -->
        <nav class="navbar justify-content-end navbar-expand-lg">
            <ul class="nav">
                <li class="nav-item"><a class="nav-link" href="/"><i
class="fa fa-home"></i> Home</a></li>
                <li class="nav-item"><a class="nav-link" href="#"><i
class="fa fa-sign-in"></i> Entrar</a></li>
                <li class="nav-item"><a class="nav-link" href="#"><i
class="fa fa-edit"></i> Registrar</a></li>
            </ul>
        </nav>
        <!-- Até aqui -->
    </header>
    {% block content %}{% endblock %}
    <!-- Adicione as classes page-footer, font-small e blue na tag footer
-->
    <footer class="page-footer font-small blue">
        <!-- Adicione o código a seguir dentro da footer -->
        <div class="footer-copyright text-center py-3">©2020 Copyright:
        <a href="https://www.casadocodigo.com.br/">
Casadocodigo.com.br</a>
        </div>
        <!-- Até aqui -->
    </footer>

    <!-- Adicione os scripts a seguir -->
    <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js">
</script>
    <script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.j
s"></script>
```

```
    <!-- Até aqui -->
    {% block scripts %}{% endblock %}
  </body>
</html>
```

Como podemos ver, os arquivos `css` e `js` que foram adicionados ficam em `cdns`. Em um projeto é recomendável salvar esses arquivos na pasta `static`, mas para aprendizado podemos usá-los pela `cdn` mesmo.

Nós também adicionamos um arquivo chamado `global.css`, onde teremos as regras que serão globais em nosso sistema. Mais à frente vamos criá-lo.

7.2 Arquivo home.html

Agora vamos criar nosso arquivo `home.html`. Crie uma pasta chamada `home` dentro do diretório `templates` e, dentro dela, crie o arquivo `home.html`.

medicSearch/templates/home/home.html

```
{% extends "base.html" %}
{% load static %}

{% block title %}Início{% endblock %}

{% block styles %}
    <link rel="stylesheet" href="{% static 'css/home.css' %}">
{% endblock %}

{% block content %}
<div id="content">Olá mundo</div>
{% endblock %}

{% block scripts %}
```

```
    <script src="{% static 'js/home.js' %}"></script>
{% endblock %}
```

Como vemos nesse arquivo `html`, usamos os blocos criados no `base.html` para trazer nosso `template` pai para dentro do arquivo `home.html`. Para carregar os arquivos que serão criados em nossa pasta `static`, basta usarmos o código `{% static 'caminho/arquivo ou apenas arquivo' %}`. Esse código chamará qualquer arquivo que esteja dentro da pasta `static`.

Agora que entendemos como funciona, vamos ver algumas imagens que podem facilitar nossa compreensão:

Exemplo de um template `html`:

Figura 7.1: Tela do template

A aplicação das partes do código no template:

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
    {% block styles %}{% endblock %}
    <title>Busca Doutor - {% block title %}{% endblock %}</title>
  </head>
  <body>
    <header></header>  1

    {% block content %}{% endblock %}

    <footer></footer>  3

    <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"></script>
    {% block scripts %}{% endblock %}
  </body>
</html>
```

Figura 7.2: Código do base.html

```
{% extends "base.html" %}
{% load static %}

{% block title %}Início{% endblock %}

{% block styles %}
    <link rel="stylesheet" href="{% static 'css/home.css' %}">
{% endblock %}

{% block content %}
Olá mundo  2
{% endblock %}

{% block scripts %}
    <script src="{% static 'js/home.js' %}"></script>
{% endblock %}
```

Figura 7.3: Código do home.html

7.3 Configurando a pasta static

Agora que aprendemos a criar nossos `templates`, vamos criar nossos arquivos estáticos. Crie uma pasta chamada `static` e, dentro dela, outra, chamada `css`. Após isso, vamos criar um arquivo chamado `global.css` e adicionar o código a seguir:

medicSearch/static/css/global.css

```
#content{
    min-height: calc(100vh - 112px);
}
.image-circle{
    width: 150px;
    height: 150px;
    border-radius: 100%;
    margin: 10px auto;
    background-size: contain;
    background-color: #ccc;
}
.btn-card {
    width: 49.5%;
    float: left;
    font-size: 0.9em;
}
.btn-card:last-child{
    margin-left: 1%;
}
.specialties{
    padding: 0;
    list-style: none;
    min-height: 46px;
}
.navigation{
    margin: 0 auto;
}
.social-container{
    display: flex;
}
.social-container span{
    display: flex;
```

```
    align-content: space-between;
}
.social-container a {
    margin: 0 1%;
    width: 50%;
    height: 40px;
    line-height: 20px;
}
.social-container a, .social-container a:hover, .social-container
a:active, .social-container a:visited{
    color: #fff
}
.social-container a:first-child {
    margin-left: 0;
}
.social-container a:last-child {
    margin-right: 0;
}

.social-container a#facebook, .social-container a#facebook:visited{
    background-color: #3b5999;
}
.social-container a#google, .social-container a#google:visited{
    background-color: #dd4b39;
}
.social-container a#facebook:hover, .social-container a#facebook:active{
    background-color: #383871;
}
.social-container a#google:hover, .social-container a#google:active{
    background-color: #9e3e2c;
}
```

Precisamos criar o arquivo `home.css` dentro desta mesma pasta, para adicionar as regras `css` que a página `home` precisa ter.

medicSearch/static/css/home.css

```
body{
    background-color: #f1f7ff;
}
```

```
#content form{
    margin-top: calc(50vh - 241px);
}
#content form button{
    width: 100%;
}
`
```

Como o objetivo deste livro não é `css` e `js`, deixarei a maior parte do código já pronta aqui e também no repositório do GitHub: https://github.com/tiagoluizrs/livro_django. Alguns capítulos à frente precisaremos mexer um pouco com `js` para implementar os `endpoints` que criaremos no capítulo sobre API no Django.

Vamos ver a estrutura do projeto antes de seguir. Repare que deixarei oculto os arquivos das demais pastas, pois o objetivo é ver se a pasta `templates` e `static` estão corretas:

```
livro_django
├──medicSearch
│       migrations
│       models
│       static
│       |       css
│       |       |       global.css
│       |       |       home.css
│       templates
│       |       home
│       |       |       home.html
│       |       base.html
│       urls
│       views
│       __init__.py
│       ... Arquivos ocultos para otimizar a página
├──medicSearchAdmin
|       ... Arquivos ocultos para otimizar a página
```

Após criar a pasta `static` e os arquivos dentro dela, talvez você precise parar o `run` do Django e rodar novamente para que ele anexe a nova pasta de arquivos estáticos em nosso projeto.

Em produção é preciso rodar o comando `python manage.py collectstatic`. Ele criará uma pasta chamada `static` no diretório principal do projeto. Isso acontece porque, como cada app tem sua pasta static, em casos em que tivermos muitos apps dentro do nosso sistema, o `collectstatic` juntará todas as pastas `statics` do projeto no diretório de administração, que em nosso caso é o `medicSearchAdmin`.

Precisamos dizer para a `view` da home renderizar esse arquivo `html`. Para isso, abra o arquivo `HomeView.py` que fica dentro do diretório `medicSearch/views` e altere-o para ficar igual ao código a seguir:

medicSearch/views/HomeView.py

```
from django.shortcuts import render

def home_view(request):
    return render(request, template_name='home/home.html', status=200)
```

Diferente do método `HttpResponse` que estávamos usando anteriormente, agora usaremos o método `render`. Ele será responsável por renderizar o template `home.html` que fica dentro da pasta `templates/home`. Perceba que no método `render` não é preciso passar o diretório `templates`, mas sim apenas os diretórios e arquivos que estão dentro da pasta `templates`. Isso ocorre pois o Django, por padrão, procura os arquivos de `template html` dentro da pasta `template`. Veja também que podemos passar informações do `request` como parâmetro, com isso podemos acessar as informações da `sessão` como os dados do usuário logado.

Se tentarmos acessar a página home já veremos nosso `html` funcionando corretamente: http://localhost:8000/.

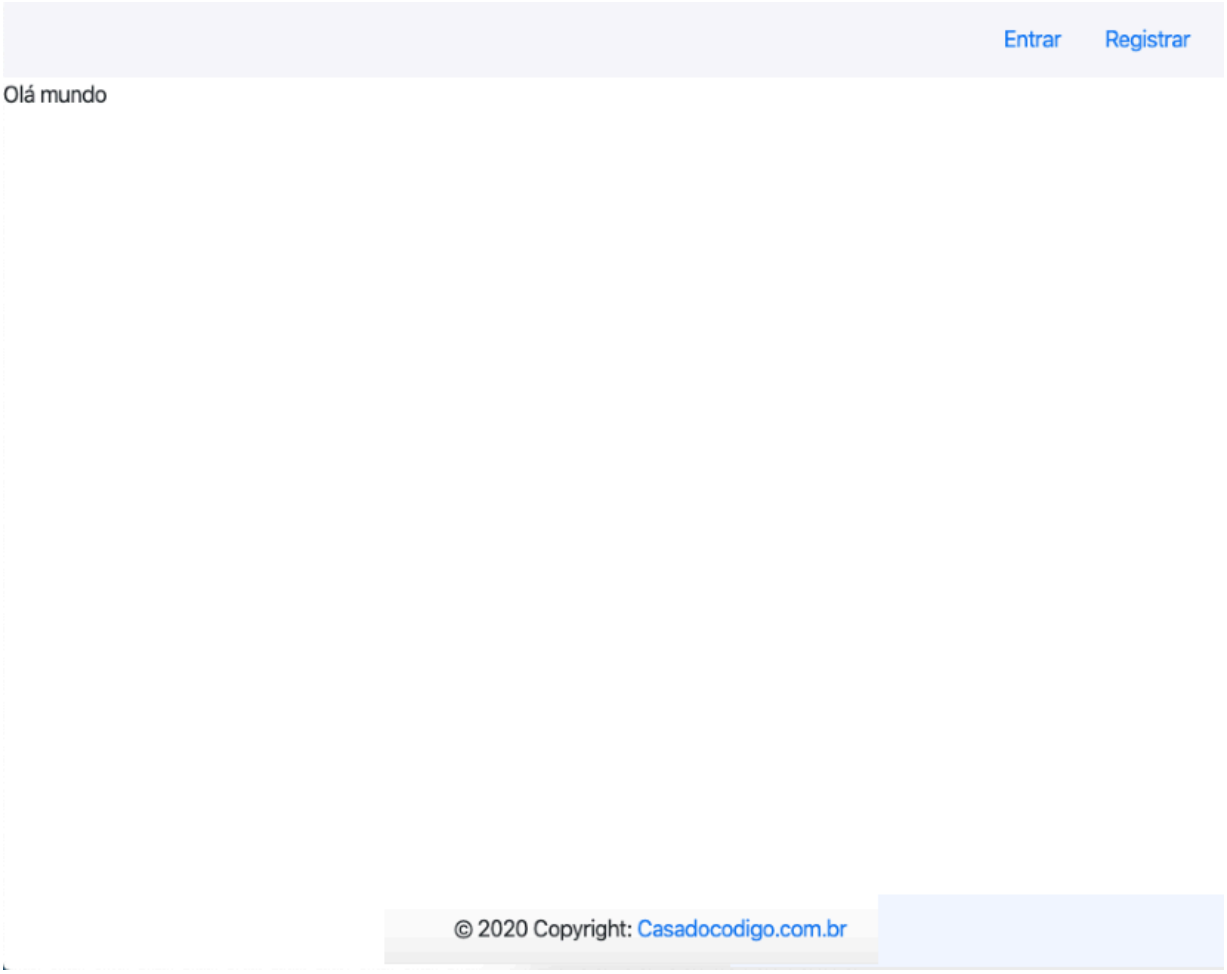


Figura 7.4: Tema home com template

Agora vamos personalizar nosso `html` da home para que possamos realizar as buscas de médicos que queremos. Altere o código do `home.html` para ficar como o código a seguir:

medicSearch/templates/home/home.html

```
{% extends "base.html" %}
{% load static %}
{% block title %}Início{% endblock %}
{% block styles %}<link rel="stylesheet" href="{% static 'css/home.css'
%}">{% endblock %}
{% block content %}
<div id="content">
    <div class="container">
```

```
      <div class="row">
        <div class="col-sm-12 col-md-6 col-lg-6 offset-sm-0 offset-md-3
offset-md-3">
            <form action="{% url 'medics' %}" method="GET">
                <h3 class="text-center">Buscar um médico</h3>
                <div class="form-row">
                    <div class="col form-group"><input type="text"
class="form-control" name="name" placeholder="Buscar por nome"></div>
                </div>
                <div class="form-row">
                    <div class="col form-group">
                        <select class="form-control" name="speciality">
                            <option value="0" selected disabled>Selecione
uma especialidade</option>
                            <option value="1">Neurocirugia</option>
                        </select>
                    </div>
                    <div class="col form-group">
                        <select class="form-control" name="state">
                            <option value="0" selected disabled>Selecione
um estado</option>
                            <option value="1">Rio de Janeiro</option>
                        </select>
                    </div>
                </div>
                <div class="form-row">
                    <div class="col form-group">
                        <select class="form-control" name="city">
                            <option value="0" selected disabled>Selecione
uma cidade</option>
                            <option value="1">Rio de Janeiro</option>
                        </select>
                    </div>
                    <div class="col form-group">
                        <select class="form-control" name="neighborhood">
                            <option value="0" selected disabled>Selecione
um bairro</option>
                            <option value="1">Botafogo</option>
                        </select>
```

```
                    </div>
                </div>
                <div class="form-row">
                    <div class="col form-group mb-0"><button type="submit"
class="btn btn-primary">Buscar</button></div>
                </div>
            </form>
        </div>
      </div>
    </div>
</div>
{% endblock %}
{% block scripts %}<script src="{% static 'js/home.js' %}"></script>{%
endblock %}
```

Ao alterar o código, teremos o formulário de busca concluído. Perceba que por enquanto nossos `selects` possuem apenas um `option`. Mais adiante traremos dinamicamente as especialidades, os estados, as cidades e os bairros que estiverem cadastrados em nossa plataforma. Ainda neste capítulo, criaremos um template que exibirá os resultados da busca de médicos que vamos fazer, mas por enquanto já podemos ver como ficou nosso formulário de buscas.

Perceba que na tag `form` utilizamos o recurso de `name` da url que aprendemos no capítulo sobre `urls` e `views` (capítulo 5). A url `/medics` possui o `name=medics` dentro do arquivo `MedicsUrl.py` (capítulo 6). Dentro do `form` foi preciso apenas chamar a tag `{% url 'medics' %}`, onde `url` é o método padrão do Django que recebe como parâmetro o nome (`name`) da url, que neste caso foi `medics`. Mais à frente veremos como criar uma url dessa utilizando o parâmetro dinâmico de id.

Ao acessar http://localhost:8000/, temos a seguinte imagem:

Figura 7.5: Tema home com busca

Perceba que o `form` possui uma `action` que leva direto para a url `http://localhost:8000/medic/` usando o verbo `GET`. Desse modo, já podemos acessar a `view` de médicos e fazer a nossa busca. O único problema é que a nossa `view` ainda não renderiza um `html` com o resultado da busca. Resolveremos isso em breve criando o `html` que exibirá o resultado dos médicos.

Dinamizando o menu

Existe algo que precisamos fazer em nosso `template html`. Perceba que os botões de entrar e registrar ficam fixos na tela e precisamos fazer com que eles apareçam apenas se não houver um usuário

logado no sistema. Caso o usuário esteja logado, precisamos mostrar o menu com o botão de sair, o botão de editar perfil e o botão de favoritos no lugar dos de entrar e registrar. Vamos alterar a `header` do nosso arquivo `base.html` :

Como vamos alterar apenas a `header` , vou exibir apenas ela aqui, mas não apague o resto do código, apenas altere a `header` para ficar igual ao código a seguir:

medicSearch/templates/base.html

```
    <header>
        <nav class="navbar justify-content-end navbar-expand-lg">
            <ul class="nav">
                <li class="nav-item"><a class="nav-link" href="/"><i
class="fa fa-home"></i> Home</a></li>
              {% if user.is_authenticated %}
                <li class="nav-item"><a class="nav-link" href="#"><i
class="fa fa-user"></i> Perfil</a></li>
                <li class="nav-item"><a class="nav-link" href="#"><i
class="fa fa-edit"></i> Editar Perfil</a></li>
                <li class="nav-item"><a class="nav-link" href="#"><i
class="fa fa-sign-out"></i> Sair</a></li>
              {% else %}
                <li class="nav-item"><a class="nav-link" href="#"><i
class="fa fa-sign-in"></i> Entrar</a></li>
                <li class="nav-item"><a class="nav-link" href="#"><i
class="fa fa-edit"></i> Registrar</a></li>
              {% endif %}
            </ul>
        </nav>
    </header>
```

Como estamos trabalhando com o `django template` , o Django nos permite acessar os dados do usuário logado através da variável `user` . Desse modo, podemos verificar se o usuário está logado usando o método `is_authenticated` que vimos no capítulo sobre `view` . Como estamos no template do Django, para usar estruturas condicionais ou loops, por exemplo, precisamos sempre abrir com o

sinal `{%` e fechar com o sinal `%}`. A seguir deixarei alguns comandos do `django template` que podem ser úteis:

Comando	O que é
`{% if condição %} {% endif %}`	Estrutura SE
`{% if condição %} {% else %} {% endif %}`	Estrutura SE SENÃO
`{% if condição %} {% elif condição %} {% else %} {% endif %}`	Estrutura SE SENÃO SE e SENÃO
`{% for i in array %} {% endfor %}`	Loop de repetição FOR

Com as alterações que fizemos, podemos ver que nosso menu já está dinâmico em http://localhost:8000/.

Figura 7.6: Tema home com template finalizada

Nossa tela home está pronta, vamos partir para a tela de resultado de médicos.

7.4 Criando o template de médicos

Agora que temos nossa home criada, vamos criar o `template` de resultado da busca de médicos. Vamos começar configurando nossa `view` de médicos para que ela renderize o `html` que vamos criar. Edite o arquivo `MedicView.py` que fica dentro da pasta `views` no diretório `medicSearch` :

medicSearch/views/MedicView.py

```
# remova o import do HttpResponse e adicione o import render igual ao do
código a seguir
from django.shortcuts import render
from medicSearch.models import Profile
from django.db.models import Q

def list_medics_view(request):
    name = request.GET.get("name")
    speciality = request.GET.get("speciality")
    neighborhood = request.GET.get("neighborhood")
    city = request.GET.get("city")
    state = request.GET.get("state")

    medics = Profile.objects
    if name is not None and name != '':
        medics = medics.filter(Q(user__first_name__contains=name) |
Q(user__username__contains=name))
    if speciality is not None:
        medics = medics.filter(specialties__id=speciality)

    if neighborhood is not None:
        medics = medics.filter(addresses__neighborhood__id=neighborhood)
    else:
        if city is not None:
            medics = medics.filter(addresses__neighborhood__city__id=city)
        elif state is not None:
            medics =
medics.filter(addresses__neighborhood__city__state__id=state)

    """Altere o código anterior que possuía um print criando agora um
dicionário chamado `context`
    e passando uma chave chamada `medics` com o valor da consulta `SQL`
que foi feita"""
    context = {
        'medics': medics
    }
    # Altere `HttpResponse` e adicione `render` no lugar igual ao código a
```

```
seguir:
    return render(request, template_name='medic/medics.html',
context=context, status=200)
```

Perceba que o método `render` possui um parâmetro chamado `context`, que é onde podemos passar um dicionário com dados que poderão ser acessados pelo nosso `template`. Veremos isso no código `medics.html` que vamos criar.

Agora que configuramos a `view`, vamos criar o arquivo `html`. Para isso, comece criando uma pasta chamada `medic` dentro do diretório de `templates` e dentro desta pasta crie um arquivo chamado `medics.html` e adicione o seguinte código:

medicSearch/templates/medic/medics.html

```
{% extends "base.html" %}
{% load static %}
{% block title %}Médicos{% endblock %}
{% block content %}
<div id="content">
    <div class="container">
        <div class="alert alert-info">Foram encontrados: {{ medics |
length }} medico(s)</div>
        <div class="row">
        {% for medic in medics %}
            <div class="col-xs-12 col-md-3 col-lg-3">
                <div class="card mb-4">
                    <div class="image-circle" style="background-image:
url('/media/{{medic.image}}');"></div>
                    <div class="card-body">
                        <h5 class="card-title">
{{medic.user.get_full_name}}</h5>
                        <a href={% url 'profile' medic.user.id %}
class="btn btn-primary btn-card">Ver médico</a>
                        <button class="btn btn-danger btn-card"><i
class="fa fa-heart"></i> Favoritos</a>
                    </div>
                </div>
```

```
            </div>
        {% endfor %}
        </div>
    </div>
</div>
{% endblock %}
```

Vamos entender o que fizemos aqui:

- **medics**: variável passada no `context` da view `MedicView` que possui uma lista de objetos do tipo `Profile` e pode ser utilizada a qualquer momento no `template`.
- **{{ medics | length }}**: aqui estamos exibindo o total de itens que a lista `medics` possui; é equivalente ao método `len()` do Python.
- **{% for medic in medics %}**: aqui temos a variável `medics` sendo iterada em um laço do tipo `for`, onde cada item será um objeto do tipo `Profile` que renderizará um card com as informações de cada médico.
- **{% url 'profile' medic.user.id %}**: aqui estamos utilizando o método padrão de url do Django que já vimos anteriormente.

> Perceba que o botão de adicionar aos favoritos já está criado. Mais à frente vamos criar a ação de salvar nos favoritos.

Precisamos fazer mais uma coisa no card do médico: exibir as especialidades dele e, para não ficar muita informação, exibiremos o primeiro endereço em que o médico atende. Os demais endereços aparecerão na tela do perfil do médico que faremos mais à frente. Vamos lá, altere o arquivo `medics.html` na parte do card. O código a seguir tem apenas o card para não ficar muito grande, ele vai mostrar apenas o que tem que ser alterado, mas não descarte o resto do código que não está sendo exibido:

medicSearch/templates/medic/medics.html

```
...código anterior oculto
<div class="col-xs-12 col-md-3 col-lg-3">
    <div class="card mb-4">
        <div class="image-circle" style="background-image:
url('/media/{{medic.image}}');"></div>
        <div class="card-body">
            <h5 class="card-title">{{medic.user.get_full_name}}</h5>
            <!-- Adicione o código a seguir após o h5 que exibe o nome do
médico -->
            <ul class="specialties">
                {% for speciality in medic.specialties.all %}
                <li>{{speciality}}</li>
                {% endfor %}
            </ul>
            <div class="address mb-2" title="
{{medic.addresses.first.address}}" alt="
{{medic.addresses.first.address}}">
                {{medic.addresses.first.address | default:"Nenhum
endereço." | slice:":15"}}...
            </div>
            <!-- Até aqui -->
            <a href={% url 'profile' medic.user.id %} class="btn btn-
primary btn-card">Ver médico</a>
            {% if user.is_authenticated %}
            <button type="submit" class="btn btn-danger btn-card"><i
class="fa fa-heart"></i> Favoritos</a>
            {% endif %}
        </div>
    </div>
</div>
...existe código abaixo que está oculto, não apague
```

Perceba que para exibir todas as especialidades atreladas ao perfil precisamos seguir um padrão do Django: para acessar um atributo `many to any`, o padrão é `objeto.atributo_many_to_many.metodo_all`, então no caso do código anterior fica `medic.specialties.all`, objeto `medic`, atributo `specialties` e método `all`. Você também pode fazer isso em uma `view` do Django, mas lá o método `all` precisa ter

parênteses, assim, `all()`. Agora temos a listagem do médico sendo exibida.

Perceba que fizemos o mesmo para o endereço, mas em vez de `all`, usamos o `first`. Dessa forma, ele retorna somente o primeiro valor `many to many` de `address` atribuído ao perfil. Assim, usamos o atributo `address` para acessar o endereço completo e, para não ficar tão grande, usamos o filtro `slice` para dizer que queremos que seja exibido apenas da primeira até a décima quinta letra. Também usamos o filtro `default` que serve para exibir um texto padrão, caso o atributo em questão tenha valor nulo ou vazio.

Nosso box está quase finalizado. Para fechar precisamos alterar nossa `view` para exibir as avaliações de cada médico. Vamos criar um método na model `Profile` que exibirá a média de avaliação do perfil do médico. Abra a model `Profile` e adicione o método a seguir:

medicSearch/models/Profile.py

```
from medicSearch.models import *
# Adicione o código a seguir
from django.db.models import Sum, Count

... Código oculto, não apague
# Aqui estarão os demais métodos e a classe Profile
... Código oculto, não apague
    @receiver(post_save, sender=User)
    def save_user_profile(sender, instance, **kwargs):
        try:
            instance.profile.save()
        except:
            pass

    # Adicione o método a seguir após o método `save_user_profile`
    def show_scoring_average(self):
        from .Rating import Rating
        try:
            ratings =
```

```
Rating.objects.filter(user_rated=self.user).aggregate(Sum('value'),
Count('user'))
            if ratings['user__count'] > 0:
                scoring_average = ratings['value__sum'] /
ratings['user__count']
                scoring_average = round(scoring_average, 2) # Arredondando
o valor para duas casas decimais
                return scoring_average
            return 'Sem avaliações'
        except:
            return 'Sem avaliações'
```

Perceba que podemos criar métodos em nossas `models` para acessá-los quando for preciso. Veja também que usamos um novo método que o ORM do Django possui chamado `aggregate`, que serve para realizarmos ações de agregação como `count` e `sum`. Nesse caso, ela vai retornar um dicionário com duas chaves que serão `value__sum` e `user__count`, pois informamos no `aggregate` que queríamos realizar o `Sum` (soma) da coluna `value` e o `Count` (contagem total de linhas) daquele `user_rated` (usuário avaliado). Perceba que foi preciso importar essas duas classes (`Sum` e `Count`) lá no topo da página. O que fizemos basicamente foi pegar a soma de todos os valores de avaliação que foram adicionados a esse médico e dividir pelo total de avaliações que existem para ele, assim temos a média para retornar a nós sempre que preciso.

Veja que também adicionamos uma pequena lógica que verifica se existem linhas de avaliação. Caso o valor dela seja 0, já sabemos que não haverá uma soma de notas e por isso esse médico não possuirá avaliação, então retornamos 'Sem avaliações'. Caso haja registros de avaliação, além de dividir a soma pelo total de avaliações, nós vamos arredondar o valor utilizando a função `round()` do Python e nesse caso arredondaremos para duas casas decimais, com `round(scoring_average, 2)`.

Agora vamos aplicar esse método em nosso `html`. Abra mais uma vez o arquivo `medics.html` e adicione a linha de código que vamos mostrar a seguir:

medicSearch/templates/medic/medics.html

```
...código anterior oculto
<div class="card-body">
    <h5 class="card-title">{{medic.user.get_full_name}}</h5>
    <!-- Adicione o código a seguir após o h5 para exibir a nota média de
avaliação do médico -->
    <h6>Nota: {{medic.show_scoring_average}} <i class="fa fa-star"></i>
</h6>
    <!-- Até aqui. Fica antes da tag `<ul class="specialties">` -->
    <ul class="specialties">
        {% for speciality in medic.specialties.all %}
        <li>{{speciality}}</li>
        {% endfor %}
    </ul>
    ...existe código abaixo que está oculto, não apague
</div>
...existe código abaixo que está oculto, não apague
```

Adicionando paginação

Com isso, nosso box está finalizado e, para finalizar a página de médicos, precisamos criar uma paginação para que haja um limite de resultados por página para otimizar seu carregamento. Vamos começar implementando o recurso de paginação em nossa `view`. Abra o arquivo `MedicView.py` e altere o código conforme a seguir:

medicSearch/views/MedicView.py

```
from django.shortcuts import render
from medicSearch.models import Profile
from django.db.models import Q
# Importe a linha a seguir
from django.core.paginator import Paginator

... Código oculto para otimizar espaço na página do livro
```

```
    if neighborhood is not None:
        medics = medics.filter(addresses__neighborhood__id=neighborhood)
    else:
        if city is not None:
            medics = medics.filter(addresses__neighborhood__city__id=city)
        elif state is not None:
            medics =
medics.filter(addresses__neighborhood__city__state__id=state)

    # Adicione o código a seguir após a finalização da consulta
    if len(medics) > 0:
        paginator = Paginator(medics, 8)
        page = request.GET.get('page')
        medics = paginator.get_page(page)

    get_copy = request.GET.copy()
    parameters = get_copy.pop('page', True) and get_copy.urlencode()

    # No dicionário `context` adicione mais uma chave chamada `parameters`
    context = {
        'medics': medics,
        'parameters': parameters
    }

    return render(request, template_name='medic/medics.html',
context=context, status=200)
```

O código anterior implementa a classe `Paginator`. Dentro dela, nós passamos o objeto da consulta e a quantidade que desejamos retornar para nosso `html`. A instância da classe `Paginator` que foi criada verificará através do método `get_page` qual página foi selecionada pelo usuário e gerará o resultado com base nela.

Um problema que teríamos com a paginação seria a perda dos parâmetros da nossa url toda vez que trocássemos de paginação. Para resolver isso, estamos criando uma variável chamada `parameters` e copiando para ela os parâmetros atuais da `url` através do `request.GET.copy()`, removendo o parâmetro `page` com o método `pop('page', True)`. Vamos usar esta variável em nosso `template` já já.

O método `get_copy.urlencode()` nos trará o seguinte resultado `page=2&page=1&name=&speciality=1&state=1&city=1&neighborhood=1`. Com isso, ao trocarmos de um número de paginação para outro não perderemos os parâmetros da `url`, pois usaremos a variável `parameters` para resolver esse problema. Veja isso a seguir.

Abra o arquivo `medics.html` e adicione o código a seguir para criar a paginação:

medicSearch/templates/medic/medics.html

```
{% extends "base.html" %}
        ... Código oculto para otimizar espaço na página do livro
        {% endfor %}
        </div>
        <!-- Adicione o código a seguir em seu html após o fechamento da
tag row que envolver o `for medic in medics`  -->
        <div class="row">
            <nav aria-label="Page navigation" class="navigation">
              <ul class="pagination">
                {% if medics.has_previous %}
                  <!-- Perceba que usamos a variável parameters sempre
para manter os parâmetros da url mesmo quando trocamos entre uma número de
paginação e outro -->
                  <li class="page-item"><a class="page-link" href="?
page=1{{ parameters }}">&laquo; Primeiro</a></li>
                    <li class="page-item"><a class="page-link" href="?
page={{ medics.previous_page_number }}{{ parameters }}">Anterior</a></li>
                {% endif %}
                  <li class="page-item"><a class="page-link"
href="#">Página {{ medics.number }} de {{ medics.paginator.num_pages }}.
</a></li>
                {% if medics.has_next %}
                    <li class="page-item"><a class="page-link" href="?
page={{ medics.next_page_number }}{{ parameters }}">Próximo</a></li>
                    <li class="page-item"><a class="page-link" href="?
page={{ medics.paginator.num_pages }}{{ parameters }}">Último &raquo;</a>
</li>
                {% endif %}
```

```
            </ul>
          </nav>
        </div>
        <!-- Até aqui -->
    </div>
</div>
{% endblock %}
```

Quando passamos a lista `medics` para dentro da classe `Paginator`, a classe gerou uma lista com o total de paginações necessárias para essa tela e nos retornou também alguns métodos que usaremos para transitar entre os resultados paginados. Veja-os a seguir:

- **has_previous**: método que verifica se existe página anterior.
- **previous_page_number**: método que vai para a página anterior.
- **has_next**: método que verifica se existe uma próxima página.
- **paginator.num_pages**: número de páginas do paginador.
- **next_page_number**: método que vai para a próxima página.

Perceba que também acessamos a variável `parameters` que criamos na `view`. Com ela, vamos concatenar os métodos da paginação para ir para a próxima e anterior, mas manteremos os parâmetros de pesquisa do médico.

Veja no `html` que ele tem os métodos de que falamos na lista anterior. Esses métodos já têm dentro de si a próxima página e a anterior, desse modo, passamos junto com os `parameters` o parâmetro `page` com os valores da próxima página e da anterior. Se não tivéssemos removido o `page` do `parameters` lá na `view`, ficariam dois parâmetros `page` nos botões de próximo e anterior.

A linha `?page={{ medics.previous_page_number }}{{ parameters }}` tem como resultado, por exemplo, `?page=2&name=&speciality=1&state=1&city=1&neighborhood=1`. Esse resultado aparece se estivermos na página 1. Se o `page` não tivesse sido removido do `parameters` teríamos algo assim: `?page=1&page=2&name=&speciality=1&state=1&city=1&neighborhood=1`.

Perceba que os métodos `has` verificam se existe próxima página e anterior; caso existam, o `html` renderiza os links que acionam os métodos `previous` e `next`.

Com tudo isso, temos a página de listagem do médico finalizada e com uma paginação. Podemos ver um exemplo de página com oito resultados por página em http://localhost:8000/medic.

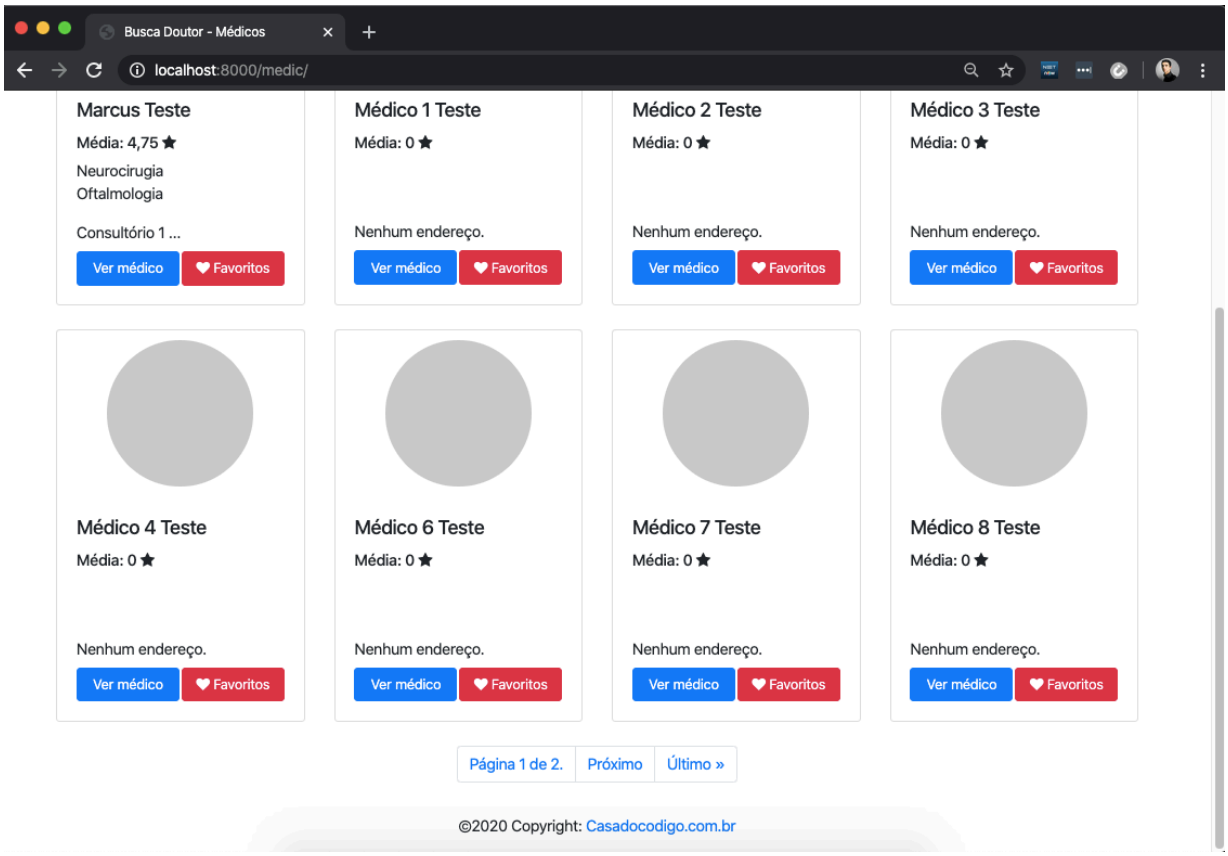


Figura 7.7: Tema médicos com template finalizado

No próximo capítulo, que será a continuação deste, aprenderemos a customizar um formulário dentro do nosso template `HTML` e criaremos o template da página de perfil da nossa plataforma.

Capítulo 8
Trabalhando com templates - Parte II

Dando continuidade ao capítulo anterior, começaremos criando um form que será customizado dentro do template da nossa tela de médicos.

8.1 Customizando um form no template

Criaremos o template que mostrará todos os médicos que estão na lista de favoritos do perfil, mas antes disso precisamos criar uma `url` que será adicionada para inserir um médico ao perfil do usuário como favorito.

Nos próximos capítulos deste livro, veremos melhor como criar um formulário através do Django, mas o que faremos aqui é um formulário `html` direto no template para aprendermos como fazer isso através do `django template`, sem usar o `django forms`. As duas maneiras que veremos no livro são úteis e você precisará escolher qual atenderá melhor à sua necessidade.

Abra o arquivo `medics.html` dentro da pasta `templates` e faça as alterações a seguir. Veja que o resto do código não aparece para não ficar enorme, mas nada foi removido, apenas adicionado:

medicSearch/templates/medic/medics.html

```
...código anterior oculto
<a href="{% url 'profile' medic.user.id %}" class="btn btn-primary btn-
card">Ver médico</a>
<!-- Altere o botão favoritos para o código a seguir. -->
{% if user.is_authenticated %}
<form method="POST" action="/medic/favorite">
    {% csrf_token %}
```

```
    <input type="hidden" value="{{medic.user.id}}" name="id">
    <input type="hidden" value="{{request.GET.page}}" name="page">
    <input type="hidden" value="{{request.GET.name}}" name="name">
    <input type="hidden" value="{{request.GET.speciality}}"
name="speciality">
    <input type="hidden" value="{{request.GET.neighborhood}}"
name="neighborhood">
    <input type="hidden" value="{{request.GET.city}}" name="city">
    <input type="hidden" value="{{request.GET.state}}" name="state">
    <button type="submit" class="btn btn-danger btn-card"><i class="fa fa-
heart"></i> Favoritos</button>
</form>
{% endif %}
<!-- Até aqui -->
...existe código abaixo que está oculto, não apague
```

> Esta não é a única maneira de criar uma ação de favoritar um médico. Poderíamos utilizar uma requisição AJAX, mas como o foco do livro não é falar sobre criação de APIs, vamos deixar dessa forma. Queremos apenas mostrar como funciona a criação de um formulário básico dentro do template.

Perceba que criamos um formulário de tipo `post`, que possui uma `action` que vai para a `view` de médicos `medic/favorite`. Veja que para acessar nossos dados da url nós usamos o `request.GET`. Toda vez que for preciso acessar um parâmetro da url é só usar `request.GET.parametro`, simples assim.

Além disso, passamos `csrf_token` entre `{% %}`, essa variável contém um token que protege a requisição contra falsificações de solicitação entre sites. A sigla `csrf` vem de um dos ataques mais conhecidos que existe, o *cross-site request forgery*. No próximo capítulo, veremos um pouco sobre ele. O importante aqui é saber que o Django, através do `csrf_token`, evita ao máximo que ocorram esses tipos de ataque que tentam fraudar uma requisição web.

Como sabemos, nada é 100% seguro, mas fique tranquilo, pois as implementações que o Django utiliza estão entre as mais seguras que existem atualmente entre os frameworks de aplicação web. Com esse `token` que passamos em nosso form, perceba que cada vez que atualizamos a página um novo token é gerado para ela, trazendo uma grande segurança para a aplicação web. Se o token não estiver na página, um erro similar ao da imagem a seguir aparecerá.

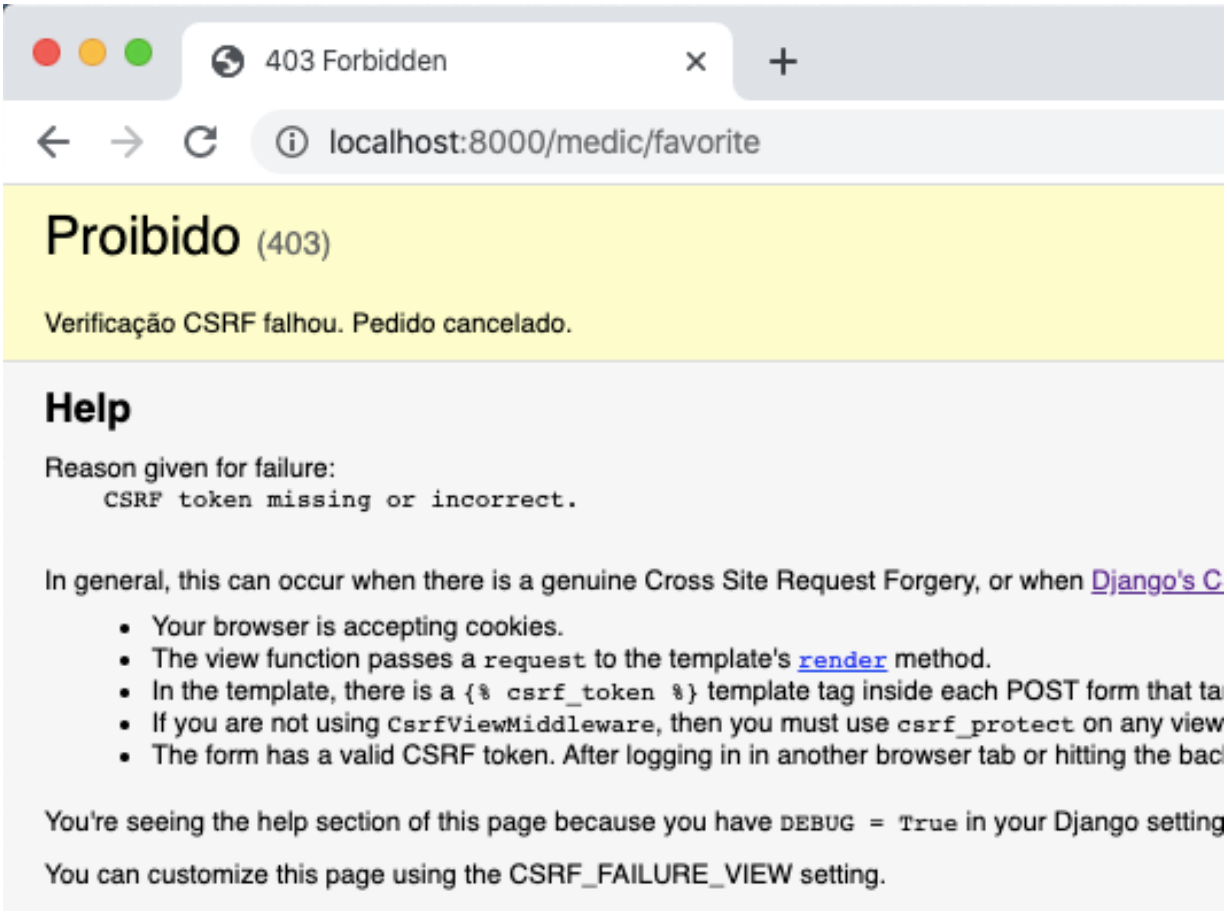


Figura 8.1: Falta de token contra fraude de requisição

Com nosso formulário de inclusão de favorito preparado no `html`, vamos abrir nossa `view` `MedicView.py` e criar um método `view` que será responsável por adicionar o médico aos favoritos do usuário logado.

`medicSearch/views/MedicView.py`

```python
# Adicione `redirect` ao import do `django.shortcuts`
from django.shortcuts import render, redirect
from medicSearch.models import Profile
from django.db.models import Q
from django.core.paginator import Paginator

def list_medics_view(request):
    # ...código oculto
    # Crie o método a seguir após o método `list_medics_view`
def add_favorite_view(request):
    page = request.POST.get("page")
    name = request.POST.get("name")
    speciality = request.POST.get("speciality")
    neighborhood = request.POST.get("neighborhood")
    city = request.POST.get("city")
    state = request.POST.get("state")
    id = request.POST.get("id")

    try:
        profile = Profile.objects.filter(user=request.user).first()
        medic = Profile.objects.filter(user__id=id).first()
        profile.favorites.add(medic.user)
        profile.save()
        msg = "Favorito adicionado com sucesso"
        _type = "success"
    except Exception as e:
        print("Erro %s" % e)
        msg = "Um erro ocorreu ao salvar o médico nos favoritos"
        _type = "danger"

    if page:
        arguments = "?page=%s" % (page)
    else:
        arguments = "?page=1"
    if name:
        arguments += "&name=%s" % name
    if speciality:
        arguments += "&specinality=%s" % speciality
    if neighborhood:
```

```
        arguments += "&neighborhood=%s" % neighborhood
    if city:
        arguments += "&city=%s" % city
    if state:
        arguments += "&state=%s" % state

    arguments += "&msg=%s&type=%s" % (msg, _type)

    return redirect(to='/medic/%s' % arguments)
```

Vamos entender o que há de novo no código:

- **request.POST.get**: é usado para pegar os dados que são enviados via requisição `POST`. O uso completo dela é `request.POST.get('nome_do_input')`.
- **redirect()**: método que usamos para redirecionar o usuário para uma `url` específica. O parâmetro principal dela é o `to`.
- **to**: parâmetro onde colocamos a `url` para redirecionar o usuário.
- **add()**: método que usamos em um atributo de tipo `many_to_many` para adicionar um objeto a ele. No caso anterior, o campo `favorites` pode receber um ou muitos `users` nele, assim, através do `add(medic.user)`, nós inserimos o usuário do médico que foi passado como favorito do usuário que está logado.

As linhas que estão dentro do `try e except` são uma consulta que pega o perfil do usuário logado através do `request.user`, atribuindo-o como um `objeto` na variável `profile`, e pega o perfil do médico pelo `id` que enviamos através da requisição `POST` e o atribui como objeto na variável `medic`.

Com isso, pegamos o objeto `profile` e acessamos seu atributo `favorites`. Como `favorites` é do tipo `many_to_many`, chamamos o método `add()` para adicionar o objeto `medic` ao `favorites` do `profile` e após isso salvamos o `profile` com o `medic` atrelado ao `favorites` dele.

Uma coisa interessante que fizemos no código foi criar duas variáveis chamadas `msg` e `_type`. Elas serão usadas para exibir uma mensagem no `template`, informando se o médico foi adicionado aos favoritos com sucesso ou se um erro ocorreu. Perceba que passamos essas variáveis como novos parâmetros da `url`. Já já vamos alterar o `html` para exibir essa mensagem.

Agora, vamos criar a `url` que vai chamar a `view` que acabamos de criar. Abra o arquivo `MedicUrls.py` dentro da pasta `urls` e adicione as linhas a seguir:

livro_django/medicSearch/urls/MedicUrls.py

```
from django.urls import path
# Import o método `add_favorite_view` na linha a seguir
from medicSearch.views.MedicView import list_medics_view,
add_favorite_view

urlpatterns = [
    path("", list_medics_view, name='medics'),
    # Adicione o novo path na linha a seguir
    path("favorite", add_favorite_view, name='medic-favorite'),
]
```

Com isso, precisamos apenas preparar a mensagem que deverá ser exibida no `html`, abra o arquivo `medics.html` dentro da pasta `templates` e adicione o código a seguir. Repare que existe código oculto, não apague nada, apenas adicione o código:

medicSearch/templates/medic/medics.html

```
...código anterior oculto
<div id="content">
    <div class="container">
        <div class="alert alert-info">Foram encontrados: {{ medics |
length }} medico(s)</div>
        <!-- Adicione o código a seguir após o alert de info da quantidade
de médicos encontrados -->
        {% if request.GET.msg %}
```

```
        <div class="alert alert-{{request.GET.type}}">{{ request.GET.msg
}}</div>
        {% endif %}
        <!-- Até aqui -->
        <div class="row">
        {% for medic in medics %}
        ...existe código abaixo que está oculto, não apague
...existe código abaixo que está oculto, não apague
```

Perceba que sempre verificamos se existe o parâmetro `msg` através da linha `{% if request.GET.msg %}`. Caso exista, ele exibirá a mensagem e o tipo dela. Se a mensagem for um erro, o `type` dela será `danger`, alterando a cor do `alert` para vermelho; se a mensagem for de sucesso, o `type` será um `success`, exibindo a mensagem em verde. Quando não houver o parâmetro `msg` na `url`, não será exibida mensagem nenhuma.

Podemos ver como ficou a ação de adicionar um médico ao favorito.

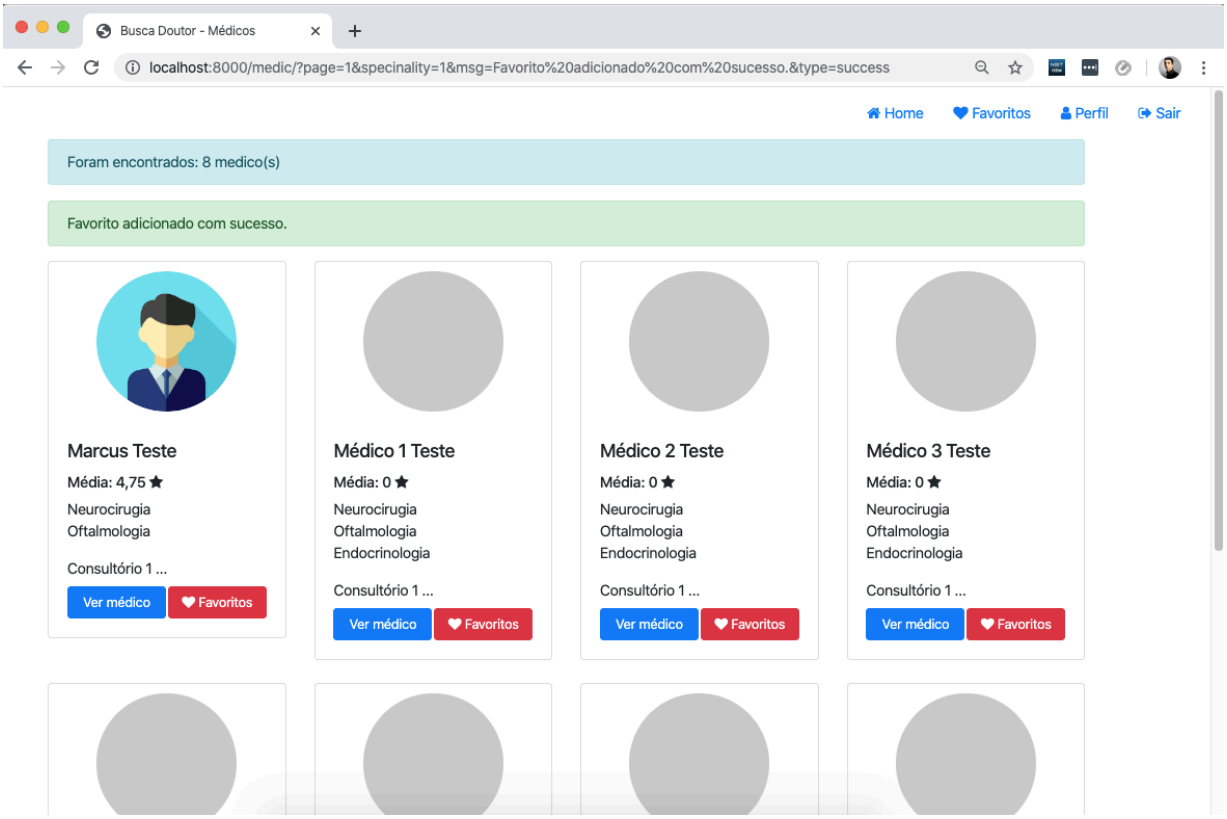


Figura 8.2: Adicionar aos favoritos

Agora que finalizamos os favoritos, para concluir ainda falta criarmos o template de perfil.

8.2 Criando o template de perfil

Nesta etapa, criaremos o template que exibirá o perfil do usuário no sistema. Caso o usuário seja um médico, serão exibidos os dados do médico e suas avaliações; caso seja um paciente, serão exibidos os dados e os médicos favoritos do usuário. Vamos começar criando o arquivo `profile.html`. Crie uma pasta chamada `profile` dentro da pasta `templates` e adicione o código a seguir:

`medicSearch/templates/profile/profile.html`

```
{% extends "base.html" %}
{% load static %}
{% block title %}Médicos{% endblock %}
{% block styles %}<link rel="stylesheet" href="{% static 'css/profile.css'
%}">{% endblock %}
{% block content %}
<div id="content">
    <div class="container">
        <div class="row">
            <div class="col-xs-12 col-md-3" id="profile-area"></div>
            <div class="col-xs-12 col-md-9" id="favorites-area"></div>
            <div class="col-xs-12 col-md-3" id="addresses-area"></div>
            <div class="col-xs-12 col-md-6" id="ratings-area"></div>
        </div>
    </div>
</div>
{% endblock %}
```

A div `profile-area` sempre será exibida, enquanto `favorites-area`, `addresses-area` e `ratings-area` serão mostradas de acordo com o perfil que está sendo exibido.

Agora, precisamos criar o arquivo `css` chamado `profile.css` dentro da pasta `css`. Crie o arquivo e adicione o código a seguir:

medicSearch/static/css/profile.css

```
#image-profile{
    width: 150px;
    height: 150px;
    background-size: contain;
    margin: 10px auto;
    background-color: #ccc;
    border-radius: 100%;
}

#badge-role{
```

```
    width: 100%;
    padding: 10px 0;
    font-size: 1em;
    float: left;
    border-radius: 0;
}

#icon-edit{
    width: 35.56px;
    height: 35.56px;
    border-radius: 0;
    float: left;
    text-align: center;
    line-height: 35.56px;
    color: #fff;
    font-weight: bold;
    background-color: #157af6;
    position: absolute;
    right: 20px;
}

ul.list-group{
    float: left;
    width: 100%;
}

ul#days{
    list-style: none;
    padding: 0;
    width: 100%;
}
```

Com nossos arquivos criados, precisamos alterar nossa `view` para que ela se comporte de forma que exiba os dados do perfil solicitado. Abra a `view ProfileView.py` para ficar conforme o código a seguir:

livro_django/medicSearch/views/ProfileView.py

```
from django.shortcuts import render, redirect
from medicSearch.models import Profile
from django.core.paginator import Paginator

def list_profile_view(request, id=None):
    profile = None
    if id is None and request.user.is_authenticated:
        profile = Profile.objects.filter(user=request.user).first()
    elif id is not None:
        profile = Profile.objects.filter(user__id=id).first()
    elif not request.user.is_authenticated:
        return redirect(to='/')

    favorites = profile.show_favorites()
    if len(favorites) > 0:
        paginator = Paginator(favorites, 8)
        page = request.GET.get('page')
        favorites = paginator.get_page(page)

    context = {
        'profile': profile,
        'favorites': favorites
    }

    return render(request, template_name='profile/profile.html',
context=context, status=200)
```

Perceba que em nosso código fizemos algumas alterações. As condicionais agora verificam se existe um id. Caso não haja, é feita uma consulta com base nos dados do usuário que está logado; caso um id seja passado, é feita uma consulta baseada neste id. No primeiro caso é feita também a verificação de `is_authenticated` para ter certeza de que o usuário está logado caso não passe nenhum id na `url`. Caso o usuário não esteja logado, se houver id aparecerá o perfil daquele id, caso não haja um id na `url` o usuário é redirecionado para a `home`.

Fique à vontade para criar uma tela para o caso de um id passado retornar usuário não existente, ou retornar o usuário para a home novamente. Vai ser ótimo para praticar o conhecimento. Aqui optei por não fazer, pois não há novidades, basicamente teríamos uma condicional para ver se o profile é `None` e, caso seja, renderizamos um `html` informando que o perfil não existe.

Veja também que criamos um paginador para a propriedade `favorites`. Para isso, foi necessário acessarmos um método que retorna todos os favoritos do perfil. Criaremos esse método a seguir. Abra o arquivo `Profile.py` que fica na pasta `models` e adicione o método adiante nele:

medicSearch/models/Profile.py

```
    def show_scoring_average(self):
        from .Rating import Rating
        try:
            ratings =
Rating.objects.filter(user_rated=self.user).aggregate(Sum('value'),
Count('user'))
            if ratings['user__count'] > 0:
                scoring_average = ratings['value__sum'] /
ratings['user__count']
                scoring_average = round(scoring_average, 2) # Arredondando
o valor para duas casas decimais
                return scoring_average
            return 'Sem avaliações'
        except:
            return 'Sem avaliações'

    # Adicione o método a seguir após o método `show_scoring_average`
    def show_favorites(self):
        ids = [result.id for result in self.favorites.all()]
        return Profile.objects.filter(user__id__in=ids)
```

Com o novo método criado e com nossa view de `perfil` modificada, vamos alterar o código do `profile.html` :

medicSearch/templates/profile/profile.html

```
...código anterior oculto
<!-- Altere os códigos a seguir, o código acima está oculto, pois não precisa ser alterado -->
        <div class="container">
            <div class="row">
                <div class="col-xs-12 col-md-3" id="profile-area">
                    <div id="image-profile" style="background-image: url('/media/{{profile.image}}');"></div>
                    {% if profile.role == 1 %}
                    <p class="badge badge-primary text-center" id="badge-role"><i class="fa fa-building"></i> Administrador</p>
                    {% elif profile.role == 2 %}
                    <p class="badge badge-primary text-center" id="badge-role"><i class="fa fa-user-md"></i> Médico</p>
                    {% else %}
                    <p class="badge badge-primary text-center" id="badge-role"><i class="fa fa-user"></i> Paciente</p>
                    {% endif %}

                    {% if profile.user.id == request.user.id %}
                    <a id="icon-edit" href="#"><i class="fa fa-edit"></i></a>
                    {% endif %}
                    <ul class="list-group">
                        <li class="list-group-item">Nome: {{profile.user.get_full_name | default:"Sem nome"}}</li>
                        <li class="list-group-item">Usuário: {{profile.user.username | default:"Sem usuário"}}</li>
                        <li class="list-group-item">E-mail: {{profile.user.email | default:"Sem e-mail"}}</li>
                        <li class="list-group-item">Nascimento: {{profile.birthday | date:'d/m/Y' | default:"Sem data"}}</li>
                        {% if profile.role == 2 %}
                        <li class="list-group-item">Nota: {{
```

```
profile.show_scoring_average }}</li>
                        <li class="list-group-item">Especialidades: {{
profile.specialties.all | join:", " }}</li>
                        {% endif %}
                    </ul>
                </div>
                {% if profile.role == 1 or profile.role == 3 %}
                <div class="col-xs-12 col-md-9" id="favorites-area"></div>
                {% else %}
                <div class="col-xs-12 col-md-3" id="addresses-area"></div>
                <div class="col-xs-12 col-md-6" id="ratings-area"></div>
                {% endif %}
            </div>
        </div>
        <!-- Altere até aqui -->
    </div>
</div>
{% endblock %}
```

Nesse momento, estamos criando a área lateral esquerda da nossa tela de perfil. Lá estamos exibindo a foto do usuário, o tipo de usuário e alguns dados, como `nome`, `email`, `username`, `nascimento`, `nota média` (se for médico) e `especialidades` (se for médico). Foram usados três filtros, um para manipular a data, `date:'d/m/Y'`, um para preencher qualquer atributo que viesse nulo ou vazio com um texto, `default:"Texto padrão"`, e um para converter a lista de especialidades em uma `string` separando cada item da lista por vírgula, `join:", "`. Também verificamos se o usuário logado e o perfil exibido eram os mesmos, caso fossem, o botão de editar perfil seria exibido: `{% if profile.user.id == request.user.id %}`. Com tudo isso, nossa tela de perfil já deve estar como a da imagem seguinte:

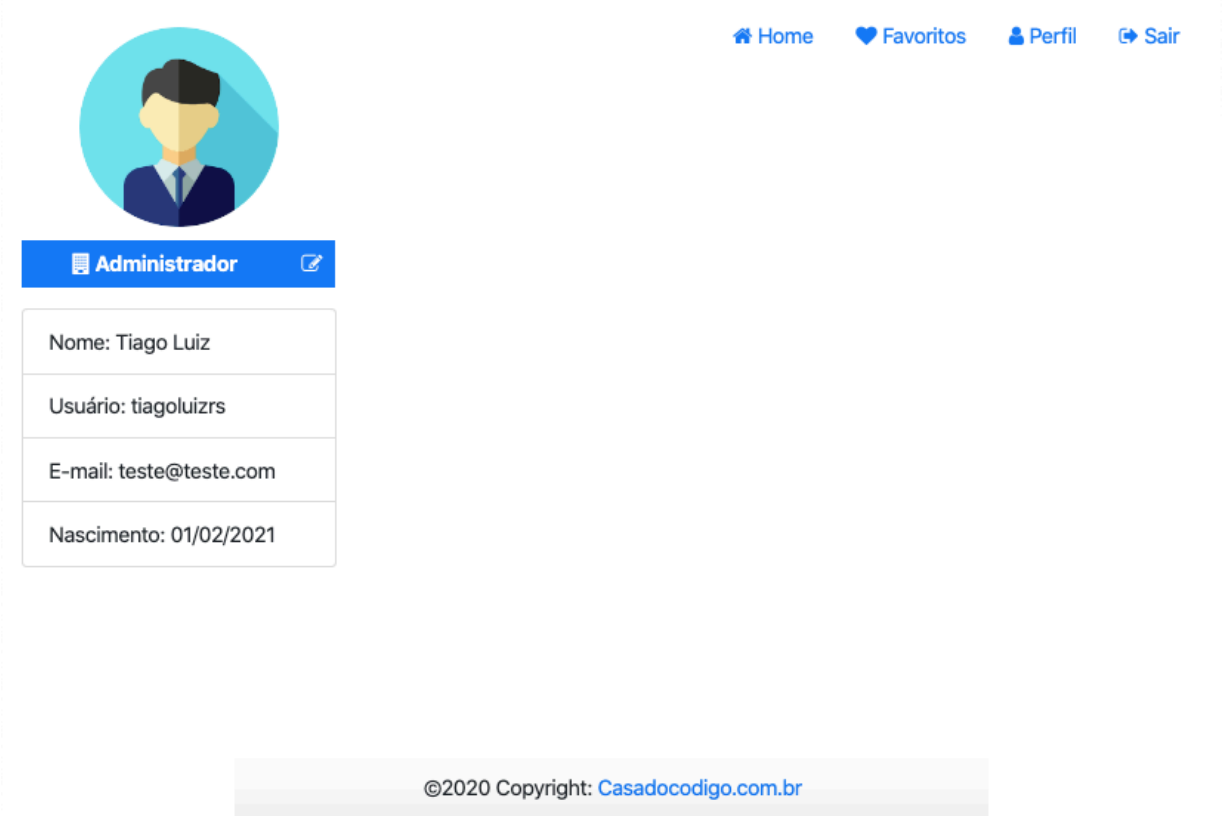


Figura 8.3: Tela de perfil em progresso

Perfil do paciente

Vamos configurar os cards de médicos favoritos para o perfil do paciente. Usaremos o mesmo card que foi utilizado para a listagem de médicos. Incluiremos o paginador também, para que possamos ter a paginação de favoritos. No lugar do botão de adicionar aos favoritos, teremos o botão de remover dos favoritos.

Vamos alterar nosso `html` adicionando o código a seguir no arquivo `Profile.html`:

`medicSearch/templates/profile/profile.html`

```
...código anterior oculto
<!-- Altere os códigos a seguir, o código acima está oculto, pois não
precisa ser alterado -->
            {% if profile.role == 1 or profile.role == 3 %}
            <div class="col-xs-12 col-md-9" id="favorites-area">
```

```
                <div class="alert alert-info">Total de favoritos:
{{favorites | length}}</div>
                <div class="row">
                    {% for favorite in favorites %}
                    <div class="col-xs-12 col-md-4 col-lg-4">
                        <div class="card mb-4">
                            <div class="image-circle" style="background-
image: url('/media/{{favorite.image}}');"></div>
                            <div class="card-body">
                                <h5 class="card-title">
{{favorite.user.get_full_name}}</h5>
                                <h6>Nota:
{{favorite.show_scoring_average}} <i class="fa fa-star"></i></h6>
                                <ul class="specialties">
                                    {% for speciality in
favorite.specialties.all %}
                                    <li>{{speciality}}</li>
                                    {% endfor %}
                                </ul>
                                <div class="address mb-2">
                                    {{favorite.addresses.first.address |
default:"Nenhum endereço." | slice:":15"}}...
                                </div>
                                <a href={% url 'profile' favorite.id %}
class="btn btn-primary btn-card">Ver médico</a>
                                {% if user.is_authenticated %}
                                <form method="POST" action={% url 'medic-
favorite-remove' %}>
                                    {% csrf_token %}
                                    <input type="hidden" value="
{{favorite.id}}" name="id">
                                    <input type="hidden" value="
{{request.GET.page}}" name="page">
                                    <button type="submit" class="btn btn-
danger btn-card"><i class="fa fa-heart"></i> Remover</button>
                                </form>
                                {% endif %}
                            </div>
                        </div>
```

```
                    </div>
                    {% endfor %}
                </div>
            </div>
            {% else %}
            <!-- Até aqui -->
            <div class="col-xs-12 col-md-3" id="addresses-area"></div>
            <div class="col-xs-12 col-md-6" id="ratings-area"></div>
            {% endif %}
        </div>
    </div>
</div>
{% endblock %}
```

> Fique tranquilo se um erro aparecer, pois chamamos o método `url` no template para o `name` `medic-favorite-remove` que pertence a uma url que ainda não criamos. Em breve vamos criá-la e o erro vai sumir.

Agora que temos os cards de favoritos, vamos adicionar a paginação ao arquivo. Altere o arquivo `profile.html` igual ao código a seguir:

medicSearch/templates/profile/profile.html

```
...código anterior oculto
                    {% for favorite in favorites %}
                    <!-- Código oculto para otimizar -->
                    {% endfor %}
                </div>
                <!-- Adicione o código a seguir após a div de class row
que envolve o for de favoritos -->
                <div class="row">
                    <nav aria-label="Page navigation" class="navigation">
                      <ul class="pagination">
                        {% if favorites.has_previous %}
                          <li class="page-item"><a class="page-link"
href="?page=1">&laquo; Primeiro</a></li>
                            <li class="page-item"><a class="page-link"
```

```
href="?page={{ favorites.previous_page_number }}">Anterior</a></li>
                        {% endif %}
                          <li class="page-item"><a class="page-link"
href="#">Página {{ favorites.number }} de {{ favorites.paginator.num_pages
}}.</a></li>
                        {% if favorites.has_next %}
                            <li class="page-item"><a class="page-link"
href="?page={{ favorites.next_page_number }}">Próximo</a></li>
                            <li class="page-item"><a class="page-link"
href="?page={{ favorites.paginator.num_pages }}">Último &raquo;</a></li>
                        {% endif %}
                      </ul>
                    </nav>
                </div>
                <!-- Até aqui -->
            </div>
            {% else %}
            <div class="col-xs-12 col-md-3" id="addresses-area"></div>
            <div class="col-xs-12 col-md-6" id="ratings-area"></div>
            {% endif %}
        </div>
    </div>
</div>
{% endblock %}
```

Sabemos que a área de favoritos do perfil é idêntica à página de resultados de busca de médicos, então não precisamos explicar o código que vimos agora. Para finalizar, vamos criar a `view` de remoção de favoritos. Para isso, abra o arquivo `MedicView.py` e vamos adicionar o método `view` a seguir em nosso código:

medicSearch/views/MedicView

```
# Crie esse método após o método add_favorite_view
def remove_favorite_view(request):
    page = request.POST.get("page")
    id = request.POST.get("id")

    try:
```

```
        profile = Profile.objects.filter(user=request.user).first()
        medic = Profile.objects.filter(user__id=id).first()
        profile.favorites.remove(medic.user)
        profile.save()
        msg = "Favorito removido com sucesso."
        _type = "success"
    except Exception as e:
        print("Erro %s" % e)
        msg = "Um erro ocorreu ao remover o médico nos favoritos."
        _type = "danger"


    if page:
        arguments = "?page=%s" % (page)
    else:
        arguments = "?page=1"

    arguments += "&msg=%s&type=%s" % (msg, _type)

    return redirect(to='/profile/%s' % arguments)
```

O método que criamos funciona de forma similar ao `add_favorite_view`, porém em vez de usarmos o `add()` para adicionar o médico ao perfil do usuário, usamos o `remove()` para remover o relacionamento `many_to_many` de `favorites` do perfil do usuário.

Agora precisamos criar a url no arquivo `MedicUrl.py`. Abra o arquivo e adicione a url e seu `name` como a do código a seguir:

medicSearch/urls/MedicUrls

```
from django.urls import path
# Import o método `remove_favorite_view` na linha a seguir
from medicSearch.views.MedicView import list_medics_view,
add_favorite_view, remove_favorite_view

urlpatterns = [
    path("", list_medics_view, name='medics'),
    path("favorite", add_favorite_view, name='medic-favorite'),
    # Adicione a linha a seguir
```

```
    path("favorite/remove", remove_favorite_view, name='medic-favorite-remove'),
]
```

Ainda precisamos adicionar uma mensagem no `profile.html` igual ao que fizemos no `medics.html`. Adicione o código a seguir no arquivo `profile.html`:

medicSearch/templates/profile/profile.html

```
...código anterior oculto
<div class="col-xs-12 col-md-9" id="favorites-area">
    <div class="alert alert-info">Total de favoritos: {{favorites | length}}</div>
    <!-- Insira o código a seguir após o alert de informação do total de favoritos do perfil -->
    {% if request.GET.msg %}
    <div class="alert alert-{{request.GET.type}}">{{ request.GET.msg }}</div>
    {% endif %}
    <!-- Até aqui -->
...existe código abaixo que está oculto, não apague
```

Pronto, o perfil de pacientes está finalizado e podemos ver o resultado final. Acesse http://localhost:8000/profile/.

Tela de perfil do paciente.

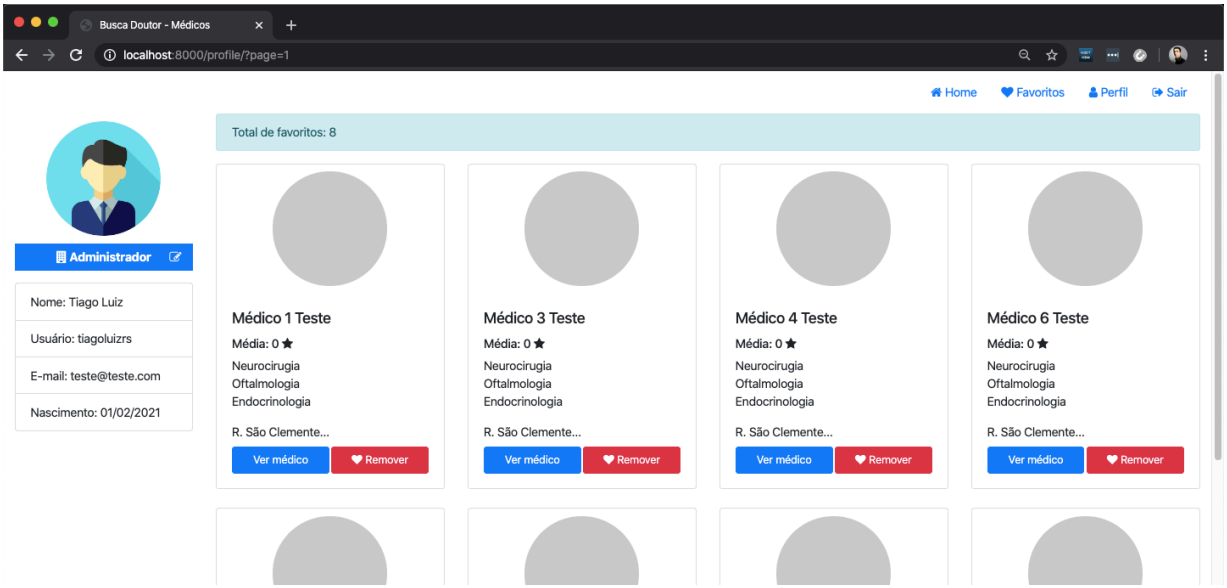


Figura 8.4: Tela de perfil do paciente

Tela de perfil do paciente - Remoção de favorito.

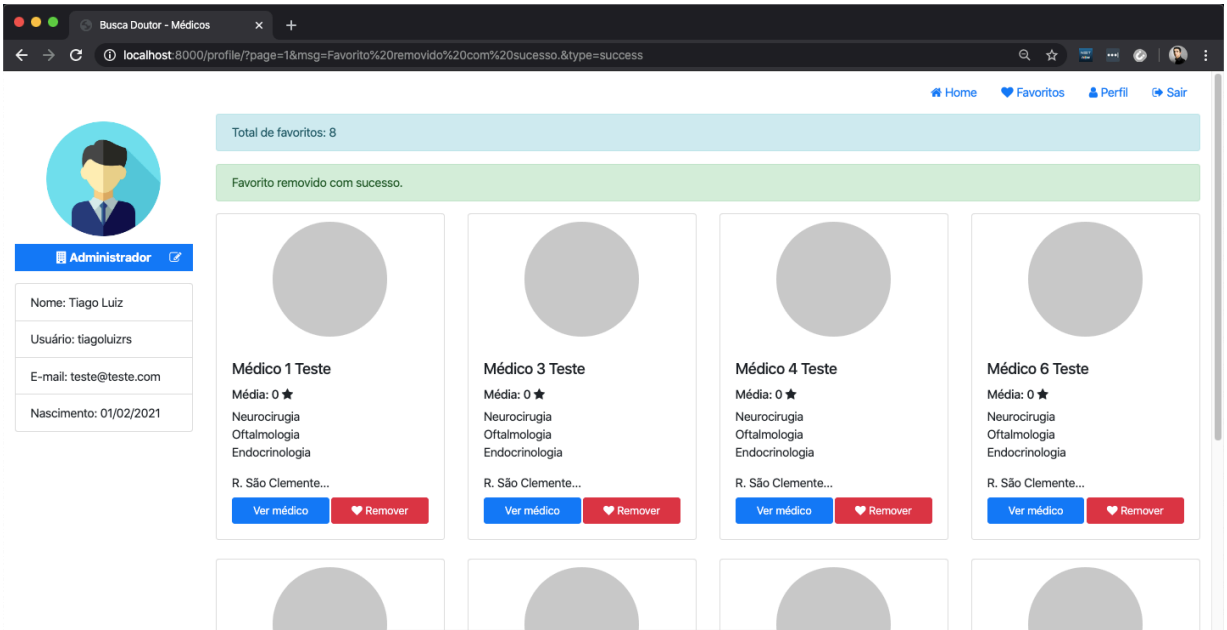


Figura 8.5: Tela de perfil do paciente

Perfil do médico

Vamos exibir as avaliações do médico e seus endereços. Vamos começar exibindo os endereços. Abra o arquivo `profile.html` e

vamos mexer dentro da tag com id `addresses-area`.

> Vale lembrar que, por enquanto, só o `admin` consegue fazer login no `sistema`. Mais à frente criaremos uma tela de login customizada para que todos os usuários façam login no sistema sem precisar ter acesso ao painel admin. Para vermos a tela como `médico`, podemos alterar nossa `role` na model de `Profile` para `médico`, só para vermos o que será feito agora. Não esqueça também que por enquanto os endereços estão sendo criados pelo admin.

medicSearch/templates/profile/profile.html

```
...código anterior oculto
<div class="col-xs-12 col-md-3" id="addresses-area">
    <div class="alert alert-info">Total de endereços:
{{profile.addresses.all | length}}</div>
    <div class="row">
        {% for address in profile.addresses.all %}
        <div class="col-xs-12 col-md-12">
            <div class="card mb-3">
                <div class="card-body">
                <h5 class="card-title">{{address.address}},
{{address.neighborhood.name}}, {{address.neighborhood.city}}</h5>
                <h6 class="card-subtitle mb-2 text-muted">Telefones:
{{address.phone}}</h6>
                <ul id="days">
                    <li>Dias de funcionamento:</li>
                    {% for day in address.days_week.all %}
                    <li>- {{day.name}} | {{address.opening_time}} -
{{address.closing_time}}</li>
                    {% endfor %}
                </ul>
                {% if profile.user.id == request.user.id %}
                <a href="/address/{{address.id}}" class="btn btn-primary">
<i class="fa fa-edit"></i></a>
                {% endif %}
```

```
                </div>
            </div>
        </div>
        {% endfor %}
    </div>
</div>
...existe código abaixo que está oculto, não apague
```

Nosso `addresses-area` é bem similar aos favoritos, nós criamos um `for` para iterar cada endereço do médico e, caso o próprio médico esteja vendo seu perfil, o botão de editar endereço será exibido. No capítulo que fala sobre `django forms` criaremos as telas de edição.

Para finalizar nosso capítulo, vamos listar as avaliações de cada médico. Deixaremos também para o capítulo de `django forms` a opção de adicionar uma avaliação ao médico. Por enquanto, só vamos exibir as avaliações existentes. Vale lembrar que, enquanto não temos como registrar avaliações, você precisará criar alguma pelo painel admin.

Vamos criar um método na `model Profile` que retornará todos os `Ratings` do perfil. Abra o arquivo `Profile.py` que fica na pasta `models` e adicione o código conforme o exemplo a seguir:

medicSearch/models/Profile.py

```
... Código anterior oculto, não o apague
    def show_favorites(self):
        ids = [result.id for result in self.favorites.all()]
        return Profile.objects.filter(user__id__in=ids)

    # Adicione o método a seguir após o método `show_favorites`
    def show_ratings(self):
        from .Rating import Rating
        return Rating.objects.filter(user_rated=self.user)
```

Esse método está pedindo para retornar todos os `ratings` em que o usuário do perfil atual foi avaliado.

Precisamos adicionar algumas linhas de código na `view` de perfil. Abra o arquivo `ProfileView.py` e altere como no código a seguir:

medicSearch/views/ProfileView.py

```
... Código anterior oculto, não o apague
    # Após os favoritos adicione o código a seguir na função
`list_profile_view`.
    # Ele é bem similar ao de favoritos
    ratings = profile.show_ratings()
    if len(ratings) > 0:
        paginator = Paginator(ratings, 8)
        page = request.GET.get('page')
        ratings = paginator.get_page(page)

    # Adicione também uma nova chave chamada `ratings`
    context = {
        'profile': profile,
        'favorites': favorites,
        'ratings': ratings
    }

    return render(request, template_name='profile/profile.html',
context=context, status=200)
```

Agora que preparamos a view para receber os `ratings` do perfil, precisamos ajeitar o `html`. Para finalizar essa etapa, abra o arquivo `profile.html` na pasta `templates` e vamos mexer na tag de id `ratings-area`:

medicSearch/templates/profile/profile.html

```
<div class="col-xs-12 col-md-6" id="ratings-area">
    <div class="alert alert-info">Total de avaliações: {{ratings |
length}}</div>
    <div class="row mb-4">
        <div class="col-xs-12 col-md-12">
            <ul class="list-group">
                {% for rating in ratings %}
                <li href="#" class="list-group-item list-group-item-
```

```
action">
                    <div class="content d-flex justify-content-between">
                        <h5 class="mb-1">{{rating.user.get_full_name}}
</h5>
                        <small>{{rating.created_at}}</small>
                    </div>
                    {{rating.opinion}}
                    <small>{{rating.value}}</small>
                    </li>
                {% endfor %}
            </ul>
        </div>
    </div>
</div>
```

Para finalizar, vamos adicionar a paginação em nosso `html`. Será idêntico ao dos favoritos. Abra o arquivo `profile.html` e adicione o código a seguir como o exemplo ensina:

medicSearch/templates/profile/profile.html

```
<div class="col-xs-12 col-md-6" id="ratings-area">
    <div class="alert alert-info">Total de avaliações: {{ratings |
length}}</div>
    <div class="row mb-4">
        ...código oculto, não apague nada
    </div>
    <!-- Adicione o código a seguir após a `row mb-4` -->
    <div class="row">
        <nav aria-label="Page navigation" class="navigation">
            <ul class="pagination">
            {% if ratings.has_previous %}
                <li class="page-item"><a class="page-link" href="?
page=1">&laquo; Primeiro</a></li>
                <li class="page-item"><a class="page-link" href="?page={{
ratings.previous_page_number }}">Anterior</a></li>
            {% endif %}
                <li class="page-item"><a class="page-link" href="#">Página
{{ ratings.number }} de {{ ratings.paginator.num_pages }}.</a></li>
            {% if ratings.has_next %}
```

```
                <li class="page-item"><a class="page-link" href="?page={{
ratings.next_page_number }}">Próximo</a></li>
                <li class="page-item"><a class="page-link" href="?page={{
ratings.paginator.num_pages }}">Último &raquo;</a></li>
            {% endif %}
            </ul>
        </nav>
    </div>
    <!-- Até aqui -->
</div>
```

Com isso, temos uma página de perfil completa. Como ainda não criamos a possibilidade de avaliar um médico, fique à vontade para criar uma avaliação pelo painel do Django para poder vê-la sendo exibida no perfil do médico. No próximo capítulo, aprenderemos a criar formulários pelo Django que nos permitirão editar nosso perfil e endereços e a enviar avaliações de médicos. Por ora, podemos ver como ficou a tela de perfil do médico:

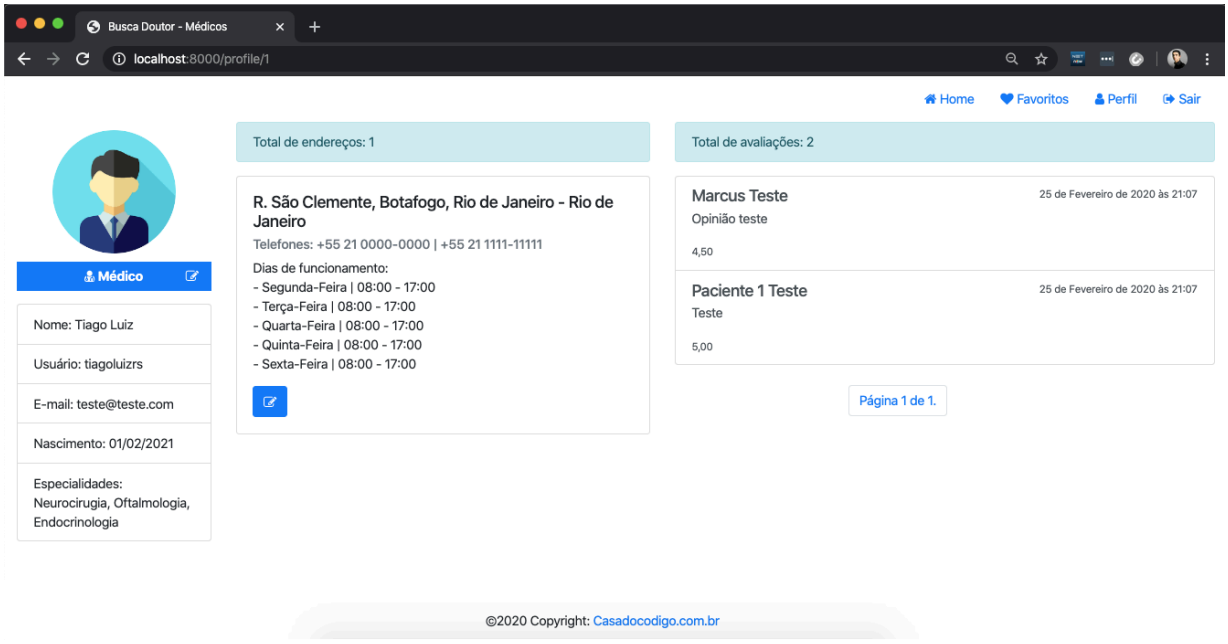


Figura 8.6: Tela de perfil do médico

CAPÍTULO 9
Trabalhando com forms - Parte I

Assim como os capítulos 7 e 8, o assunto sobre formulários também é um pouco mais extenso, por isso ele também será dividido em duas partes, o capítulo atual (9) e o capítulo 10.

Simplicidade, agilidade e limpeza são algumas das características principais do Django que o fazem ser tão utilizado no mercado de trabalho. Neste capítulo, veremos como é simples criar um formulário no Django que refletirá a estrutura de nossas models.

Nosso objetivo com esses formulários é implementá-los em nossos templates customizados. Gerá-los é muito simples e o resultado que eles trazem é excelente. Podemos criar um formulário de edição de perfil idêntico ao que fica no `Admin` do Django adicionando restrições que acharmos necessárias e com isso implementá-lo em nosso `html` com a aparência que quisermos. Vamos lá.

> Não se esqueça de fazer login na plataforma admin. Mais à frente vamos personalizar nossa tela de login para deixar a plataforma mais com a cara do sistema.

9.1 Criando um model form

Nesta etapa, precisamos criar um formulário que corresponderá à nossa tabela de profile do sistema. Formulários no Django podem ou não ser reflexos de uma model, desse modo, podemos criar formulários que serão gerados com base em uma model do nosso sistema ou apenas um formulário como o de login, que não precisa

necessariamente ser baseado em uma classe model do sistema, a escolha fica a cargo de quem está criando a model.

No caso do formulário de perfil, usaremos a model `Profile` como base. A vantagem de um form ser criado com base em uma model é que podemos utilizar o método `.save()` do `django form` para salvar ou atualizar os dados do banco de dados que são baseados nela. Veremos isso a seguir.

Dentro da pasta do nosso app `medicSearch` crie uma pasta chamada `forms` e, dentro dela, crie um arquivo chamado `UserProfileForm.py`. Vamos adicionar o seguinte código a esse arquivo:

medicSearch/forms/UserProfileForm.py

```
from django.forms import ModelForm
from django import forms
from medicSearch.models.Profile import Profile

class UserProfileForm(ModelForm):


    class Meta:
        model = Profile
        fields = ['user', 'role', 'birthday', 'image']
        # Usamos '__all__' para exibir todos os campos como itens do
formulário
        # fields = '__all__'
        # Usamos uma lista para definir quando queremos exibir campos
específicos
        # fields = ['pub_date', 'headline', 'content', 'reporter']
        # Usamos exclude para excluir campos específicos do sistema
        # exclude = ['']
        widgets = {
            'user': forms.HiddenInput(),
            'role': forms.Select(attrs={'class': "form-control"}),
            'birthday': forms.DateInput(attrs={'class': "form-control",
"type": "date"}),
```

```
            'image': forms.FileInput(attrs={'class': "form-control"})
        }
```

Vamos entender o que fizemos no código anterior:

- **from django.forms import ModelForm**: objeto que nossa classe `ProfileForm` precisa herdar para que possamos criar um formulário do Django que se baseie em uma model da nossa aplicação.
- **from django import forms**: módulo do Django que possui diversas classes que podem modificar a estrutura de um campo do formulário. Podemos colocar campos como `readonly`, `hidden`, trocar os tipos de dados nativos dos campos, adicionar máscaras de textos neles, e à frente veremos uma tabela com as classes que podemos usar para manipular nossos campos. Esse campo também é usado quando queremos criar um form que não é baseado em uma model da nossa aplicação.
- **Meta**: classe que usamos para configurar o nosso formulário que é baseado em uma model do sistema. Não é necessário quando criamos um formulário customizado que não usa uma model como base.
- **fields**: usamos esse atributo recebendo uma `lista` correspondente aos campos que desejamos que sejam exibidos em nosso formulário. Podemos passar uma `string` com `'__all__'` para dizer que queremos que todos os campos sejam exibidos no formulário.
- **exclude**: usamos esse atributo recebendo uma lista dos campos que desejamos ocultar em nosso formulário. Não podemos usar o `'__all__'` nele, pois precisamos de pelo menos um campo exibido no formulário e não pode ser usado junto com o `fields`. Ou colocamos o `fields` ou colocamos o `exclude`, não podemos usar os dois, um anula o outro.
- **widgets**: nesse atributo, como um dicionário onde podemos passar o nome do campo que desejamos modificar, podemos mudar o tipo de campo, de um `input text` para e-mail, por exemplo, ou podemos passar um `input text` para `textarea`,

podemos criar máscaras para uma data, definir um `disabled` para o `input`, `required` etc. Mais à frente veremos exemplos isolados do uso do atributo `widget`.

- **HiddenInput()**: uma das muitas classes que tem no módulo `forms` que podemos usar para modificar um campo do formulário para se comportar como um `hidden input`.
- **CheckboxInput()**: uma das muitas classes que tem no módulo `forms` que podemos usar para modificar um campo do formulário para se comportar como uma checkbox.
- **Select()**: uma das muitas classes que tem no módulo `forms` que podemos usar para modificar um campo do formulário para se comportar como um campo `select`.
- **DateInput()**: uma das muitas classes que tem no módulo `forms` que podemos usar para modificar um campo do formulário para se comportar como um campo de tipo date.
- **FileInput()**: uma das muitas classes que tem no módulo `forms` que podemos usar para modificar um campo do formulário para se comportar como um campo de tipo file.
- **attr()**: podemos passar `attrs` recebendo um dicionário com vários recursos, como `class`, `id`, `style`, `type` e qualquer outro atributo html que nosso campo precise receber. Nesses casos, adicionamos a classe do framework `bootstrap` chamada `form-control` e no `input birthday` inserimos também o atributo `type` para que o `input` seja do tipo `date` e apareça um calendário no `input` de aniversário.

Perceba que, no caso do nosso código, não mudamos o comportamento de nenhum campo, todos estão em suas formas nativas. Só criamos os widgets para os campos `role`, `birthday` e `image` para podermos adicionar uma `classe css` a eles, pois usaremos essa classe para personalizar nosso template `html`.

Como podemos perceber com esses recursos, já podemos criar um form baseado em nossa model `Profile`. Vamos ver a seguir alguns exemplos de uso do atributo `widgets` que podem ser úteis:

```
widgets = {
    'description': forms.Textarea(attrs={'cols': 80, 'rows': 20,
'required': True}),
    'password': forms.PasswordInput(),
    'date': forms.DateInput(attrs={"type": "date"}),
    'email': EmailInput(),
    'image': forms.FileInput(attrs={'class': "form-control"})
}
```

Esses são alguns exemplos que podemos fazer para alterar um campo de texto, e-mail etc. para se comportar de uma maneira diferente de sua maneira nativa.

Gerenciando campos do formulário

Algo que não queremos que aconteça é um campo, como `role` ser acessível por um usuário de tipo médico ou paciente, afinal se você é um paciente, não pode se alterar para médico ou admin. Para isso, vamos adicionar o código a seguir em nosso formulário de edição de perfil. Abra o arquivo `UserProfileForm.py` dentro da pasta `form` e adicione o bloco de código do exemplo a seguir:

medicSearch/forms/UserProfileForm.py

```
from django.forms import ModelForm
from django import forms
from medicSearch.models.Profile import Profile

class UserProfileForm(ModelForm):
    # Adicione as linhas a seguir
    def __init__(self, *args, **kwargs):
        super(UserProfileForm, self).__init__(*args, **kwargs)
        if self.instance and self.instance.role != 1:
            del self.fields['role']
    # Adicione antes da class Meta

    class Meta:
        model = Profile
```

```
        fields = ('user', 'role', 'birthday', 'image',)
        # Código oculto. Não o apague
```

O que cada linha faz:

- **self.instance**: a classe `ModelForm` tem um atributo dentro dela que retorna a `instance` atual do formulário, que nesse caso é o próprio perfil. Sendo a `instance` o formulário do próprio perfil, eu consigo acessar o atributo `role` e, caso ele seja diferente de 1, eu deletarei o field `role` para não ser renderizado no `html`.
- **del self.fields['role']**: ação que usaremos para deletar o field `role` desse formulário no momento da renderização da página caso o usuário não seja o admin.

Sabemos que o método `__init__` é o construtor de uma classe no Python. Desse modo, quando criamos a instância da classe `UserProfileForm` ele executará o método construtor e fará o que explicamos no parágrafo anterior.

9.2 Integrando nosso form a nossa view

Abra o arquivo `ProfileView.py` que fica dentro da pasta `medicSearch/views` e vamos criar uma `def` chamada `edit_profile`. Nela, vamos adicionar a lógica da nossa `view` que vai integrar nosso formulário para exibi-lo em nosso template `html`:

medicSearch/views/ProfileView.py

```
from django.core.paginator import Paginator
# Adicione o import do método get_object_or_404 a seguir
from django.shortcuts import render, redirect, get_object_or_404
from medicSearch.models import Profile
# Importe a classe UserProfileForm a seguir
from medicSearch.forms.UserProfileForm import UserProfileForm
```

```
def list_profile_view(request, id=None):
    # Código oculto. Adicione após a def list_profile_view a def
edit_profile

def edit_profile(request):
    profile = get_object_or_404(Profile, user=request.user)
    profileForm = UserProfileForm(instance=profile)

    context = {
        'profileForm': profileForm
    }

    return render(request, template_name='user/profile.html',
context=context, status=200)
```

Vamos entender o código anterior:

- **get_object_or_404**: usamos esse método para fazer uma consulta no Django. O primeiro parâmetro é a model que queremos consultar e o segundo é o campo da model que vamos consultar. Caso retorne um resultado do banco de dados, teremos como retorno uma instância da model, caso não exista um resultado, retornamos um erro de `404` para a página. Usamos isso aqui para podermos recarregar os valores do `Profile` de usuário que está logado. No caso, esse profile é nosso mesmo, com isso pegamos a instância da classe `Profile` com os valores registrados no banco e poderemos usá-lo a seguir passando-o como parâmetro para nosso `django form` .
- **UserProfileForm**: classe que criamos que será a representação da nossa model em formato de formulário em nosso template `htmlt` .
- **instance**: esse parâmetro da nossa classe de form receberá a instância da classe que queremos editar. Se a instância for `None` ele entenderá que uma nova linha deverá ser criada no banco de dados; se a instância for um objeto de Profile preenchido, passaremos esses valores para o formulário, para que possamos iniciar o formulário com os valores atuais do banco de dados. Desse modo, quando vamos editar algo, podemos fazê-lo

sabendo o que está sendo alterado, em vez de editarmos um formulário em branco sem saber o valor anterior que existiam naqueles campos.

Com isso, já podemos exibir nosso formulário no template. Como nosso formulário ainda não é um formulário de criação de usuário, mas sim de edição, vamos carregá-lo com valores predefinidos que poderão ser alterados. Mais à frente veremos como salvar novos valores. Agora veremos primeiro como exibir o formulário no template.

9.3 Criando a url da nossa view

Vamos criar a url que acessará a view da nossa edição de perfil. Crie um arquivo chamado `ProfileUrls.py` dentro da pasta `medicSearch/urls` e adicione a rota conforme o código a seguir:

medicSearch/urls/ProfileUrls.py

```
from django.urls import path
# Adicione o método edit_profile ao import a seguir
from medicSearch.views.ProfileView import list_profile_view, edit_profile

urlpatterns = [
    path("", list_profile_view, name='profiles'),
    path("<int:id>", list_profile_view, name='profile'),
    # Adicione a linha a seguir
    path("edit", edit_profile, name='edit-profile'),
]
```

Com isso, podemos partir para a criação do nosso formulário em nosso template `html` . Vamos lá.

9.4 Template de perfil

Nessa parte, vamos criar o arquivo `html` que receberá o nosso `django form`. Dentro da pasta `medicSearch/templates`, crie uma pasta chamada `user` e, dentro dela, um arquivo chamado `profile.html`.

Dentro desse `profile`, vamos primeiro adicionar a base do template, que são os menus e a tag form:

medicSearch/templates/user/profile.html

```
{% extends "base.html" %}
{% load static %}
{% block title %}Perfil{% endblock %}
{% block styles %}<link rel="stylesheet" href="{% static 'css/home.css'
%}">{% endblock %}
{% block content %}
<div id="content">
    <div class="container-fluid">
        <div class="row">
            <form class="col-md-6 col-lg-4 offset-md-3 offset-lg-4"
method="POST" action="">

            </form>
        </div>
    </div>
</div>
{% endblock %}
```

> Vamos usar o `css` da home nesta página, porque queremos ter a mesma aparência que a da página home.

Agora que temos nossa estrutura, vamos fazer tudo dentro de `<form> </form>`. Vou ocultar o código que está em volta para deixá-lo mais

curto, então fique atento e não remova o código, ele só vai estar oculto no livro.

CSRF

A seguir vamos implementar um modelo de segurança em nosso formulário e entender como ele funciona. Para isso, precisamos entender o que é o CSRF.

O *cross-site request forgery* (CSRF) é uma das vulnerabilidades mais conhecidas no mundo da web. Quando temos um formulário no sistema, se ao mesmo tempo entramos em um site malicioso podemos acabar permitindo que esse site malicioso faça uma requisição forjada ao nosso sistema sem percebermos. Ele faz isso porque conseguirá saber qual a url que estamos requisitando ao servidor.

É assustadora essa história, sem dúvidas. Para inibir esse tipo de ataque, o Django e muitos outros frameworks web utilizam um token de autenticação dentro do form que é criado pelo sistema, baseado no login do usuário e sua sessão. Como os navegadores implementam um Same Origin Policy (SOP), a janela maliciosa não conseguirá requisitar para nosso servidor o token `CSRF` que nossa aplicação cria para nossos formulários. Como nosso servidor só aceitará requisições que tenham esse token dentro do formulário, teremos uma segurança maior contra esse tipo de fraude, já que o site malicioso pode até saber a url que deve requisitar, mas se ele não envia para o servidor o token `CSRF` o Django rejeitará essa requisição.

Vamos ver a seguir como implementar nosso formulário utilizando o token `CSRF` .

Voltando ao form

Agora que sabemos o que é `CSRF` , podemos voltar à construção do formulário do Django em nosso template `html` . Abra o arquivo

`profile.html` novamente e vamos começar:

medicSearch/templates/user/profile.html

```
... código oculto
<form class="col-md-6 col-lg-6 offset-md-3 offset-lg-3" method="POST"
action="">
    <div class="row">
        {% csrf_token %}
        {% for f in profileForm %}
            {% if not f.is_hidden %}
            <div class="form-group col-md-6">
                {{ f.label }}
                {{ f }}
            </div>
            {% else %}
                {{ f }}
            {% endif %}
        {% endfor %}
    </div>
</form>
... código oculto
```

Vamos entender o que fizemos nesse código.

- **{% csrf_token %}**: aqui temos a tag padrão do token `CSRF` que é necessária para que o formulário funcione corretamente. Ela vai gerar o token de segurança do formulário. Um pequeno exemplo do token que ele gera:

```
<input type="hidden" name="csrfmiddlewaretoken"
value="izMJtRFaSc4LpBgt78ASOfVj06w0eXiTH0ImJio2mm3NNVrhT8SpuBEiKSqMO2rW">
```

- **f**: aqui temos a iteração do objeto `form`, que possui uma lista com todos os campos. Passamos form no laço `for` e conseguimos pegar cada campo do formulário. Quando usamos a tag `{{f}}`, estamos chamando o próprio campo `input`, `select` ou `textarea` referente àquele campo.

- **f.label**: esse é o método que retorna o nome da label do nosso campo. Assim podemos exibir uma label que represente aquele campo no formulário.
- **f.is_hidden**: o método `is_hidden` retorna se o `input` é do tipo `hidden` ; caso seja, retorna `True` , do contrário, retorna `False` .

Uma observação interessante é que podemos também chamar os campos por nome, vamos ver um exemplo. Não precisa implementar esse código, ele é apenas uma alternativa:

medicSearch/templates/user/profile.html

```
... código oculto
<form class="col-md-6 col-lg-6 offset-md-3 offset-lg-3" method="POST"
action="">
    <div class="row">
        {% csrf_token %}
        <div class="form-group">
            {{ profileForm.role.label }}
            {{ profileForm.role }}
        </div>
        <div class="form-group">
            {{ profileForm.birthday.label }}
            {{ profileForm.birthday }}
        </div>
        <div class="form-group">
            {{ profileForm.image.label }}
            {{ profileForm.image }}
        </div>
    </div>
</form>
... código oculto
```

Perceba que, com esse código, nossa tela de formulário já ficará da seguinte maneira, acesse pelo link `http://localhost:8000/profile/edit` :

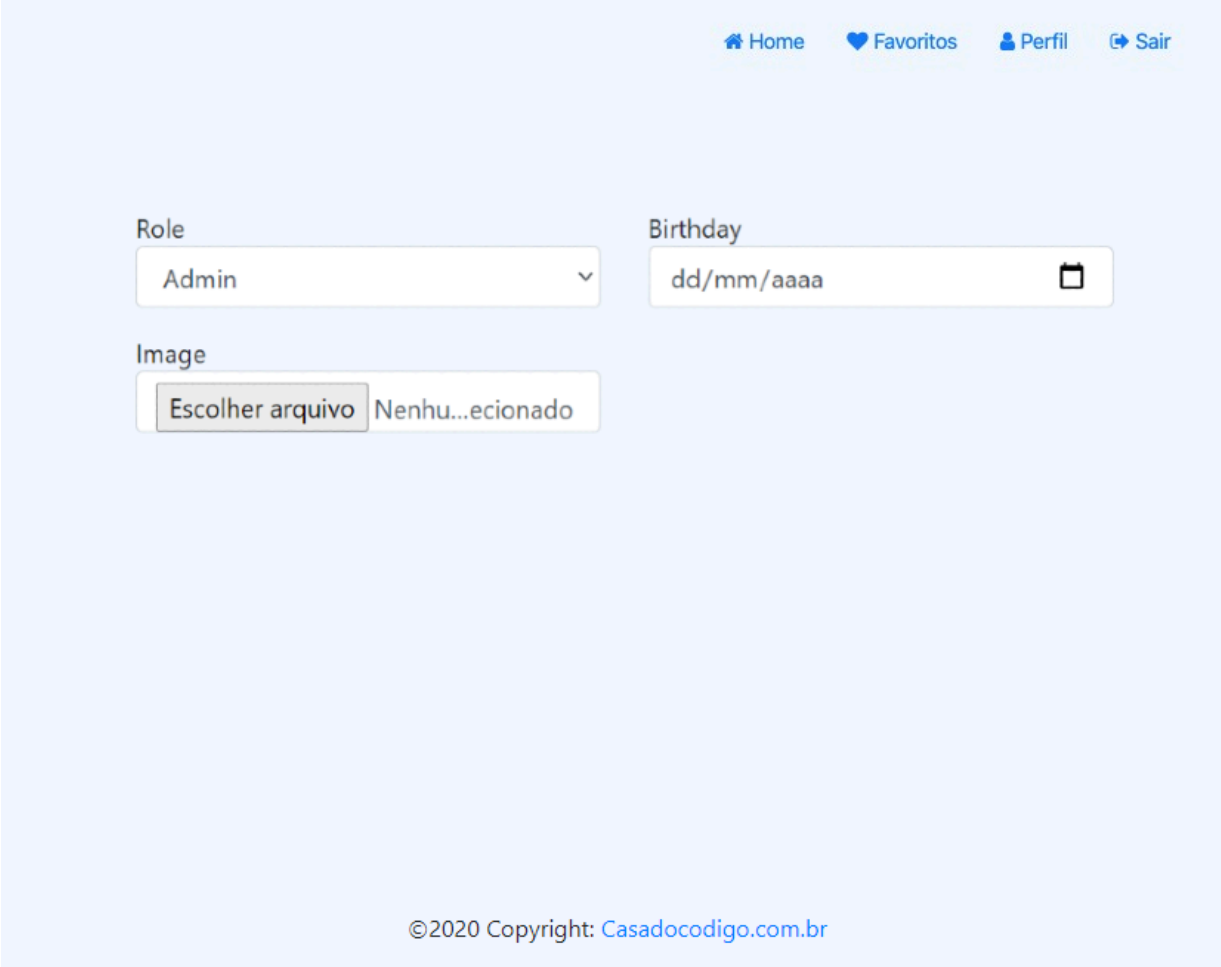


Figura 9.1: Formulário de edição de perfil parte 1

Nosso formulário ainda não tem os campos de `username`, `email`, `first_name`, `last_name` e outros. Isso ocorre porque estamos editando a model `perfil` e não a model `User`, que é uma model nativa do Django. Vamos criar uma classe de form para a tabela do usuário e adicioná-la em nosso html, e para isso precisamos adicionar algumas coisas.

Vamos abrir o arquivo `UserProfileForm.py` dentro da pasta `medicSearch/forms` e adicionar uma `class` de form para o formulário de usuário que contém os campos `username`, `email`, `first_name` e `last_name`:

medicSearch/forms/UserProfileForm.py

```
from django.forms import ModelForm
from django import forms
from medicSearch.models.Profile import Profile
# Adicione a linha a seguir no import
from django.contrib.auth.models import User

class UserProfileForm(ModelForm):
... Código ocultado

# Adicione o código a seguir após a classe UserProfileForm. Não apague a
classe UserProfileForm

class UserForm(ModelForm):
    class Meta:
        model = User
        fields = ['username', 'email', 'first_name', 'last_name']
        widgets = {
            'username': forms.TextInput(attrs={'class': "form-control"}),
            'email': forms.EmailInput(attrs={'class': "form-control"}),
            'first_name': forms.TextInput(attrs={'class': "form-
control"}),
            'last_name': forms.TextInput(attrs={'class': "form-control"})
        }
```

Acabamos de criar um formulário para editar as partes do usuário que ficam na tabela padrão do Django. Agora vamos adicionar esse form na `medicSearch/views` e no template. Comece abrindo o arquivo `ProfileView.py` dentro da pasta `medicSearch/views`:

medicSearch/views/ProfileView.py

```
from django.core.paginator import Paginator
from django.shortcuts import render, redirect, get_object_or_404
from medicSearch.models import Profile
# Adicione a classe `UserForm` na importação a seguir
from medicSearch.forms.UserProfileForm import UserProfileForm, UserForm

def list_profile_view(request, id=None):
    # Código oculto, pois não é necessário aqui
```

```
def edit_profile(request):
    profile = get_object_or_404(Profile, user=request.user)
    profileForm = UserProfileForm(instance=profile)
    # Crie uma instância da classe UserForm
    userForm = UserForm(instance=request.user)

    # Adicione uma chave `userForm` no dicionário de context passando a
instância do form
    context = {
        'profileForm': profileForm,
        'userForm': userForm
    }

    return render(request, template_name='user/profile.html',
context=context, status=200)
```

Agora vamos abrir o template e adicionar o form de usuário nele. Abra o arquivo `profile.html` da pasta `user` e adicione as linhas como no exemplo a seguir:

medicSearch/templates/user/profile.html

```
... código oculto
<form class="col-md-6 col-lg-6 offset-md-3 offset-lg-3" method="POST"
action="">
    <div class="row">
        {% csrf_token %}
        <!-- Adicione o userForm antes do profileForm -->
        {% for f in userForm %}
            {% if not f.is_hidden %}
            <div class="form-group col-md-6">
                {{ f.label }}
                {{ f }}
            </div>
            {% else %}
                {{ f }}
            {% endif %}
        {% endfor %}
        <!-- Até aqui -->
        {% for f in profileForm %}
```

```
            {% if not f.is_hidden %}
            <div class="form-group col-md-6">
                {{ f.label }}
                {{ f }}
            </div>
            {% else %}
                {{ f }}
            {% endif %}
        {% endfor %}
    </div>
</form>
... código oculto
```

Temos uma tela parecida com a tela a seguir:

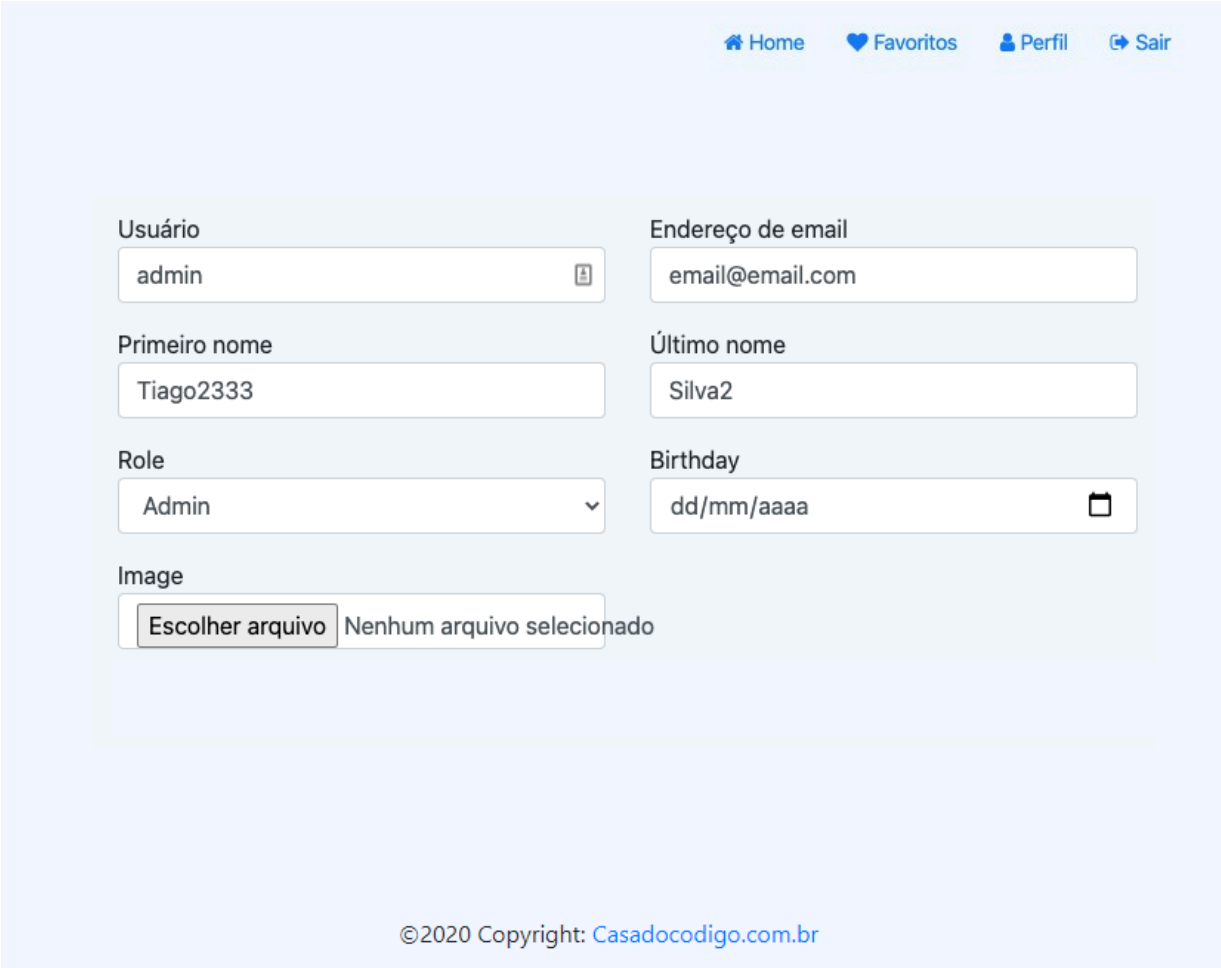


Figura 9.2: Formulário de edição de perfil completo

Nossa edição de perfil já está ganhando forma. Para fechar a edição de perfil, precisamos configurar alguns itens, como mensagem de validação e de sucesso ou erro do salvamento, botão de enviar formulário e também precisamos persistir esses dados no banco. Vamos adicionar primeiro o botão de enviar. Altere o arquivo `html` como no exemplo a seguir:

medicSearch/templates/user/profile.html

```
... código oculto
            <form class="col-md-6 col-lg-6 offset-md-3 offset-lg-3"
method="POST" action="">
                <div class="row">
                        ... código oculto, não apague
                </div>
                <!-- Adicione depois da div de class row do form -->
                <button type="submit" class="btn btn-
primary">Salvar</button>
            </form>
... código oculto
```

Agora temos um botão de envio do formulário e com isso podemos passar a salvar o que for enviado pela requisição. Para isso, abra o arquivo `ProfileView.py` dentro da pasta `medicSearch/views` e altere conforme no exemplo:

medicSearch/views/ProfileView.py

```
from django.core.paginator import Paginator
from django.shortcuts import render, redirect, get_object_or_404
from medicSearch.models import Profile
from medicSearch.forms.UserProfileForm import UserProfileForm, UserForm

def list_profile_view(request, id=None):
    # Código oculto

def edit_profile(request):
    profile = get_object_or_404(Profile, user=request.user)
```

```
    # Adicione as linhas a seguir (Perceba que os forms desceram e estão
dentro do if e do else agora)
    emailUnused = True
    if request.method == 'POST':
        profileForm = UserProfileForm(request.POST, instance=profile)
        userForm = UserForm(request.POST, instance=request.user)

        # Verifica se o e-mail que o usuário está tentando utilizar em seu
perfil já existe em outro perfil
        verifyEmail =
Profile.objects.filter(user__email=request.POST['email']).exclude(user__id
=request.user.id).first()
        emailUnused = verifyEmail is None
    else:
        profileForm = UserProfileForm(instance=profile)
        userForm = UserForm(instance=request.user)

    if profileForm.is_valid() and userForm.is_valid() and emailUnused:
        profileForm.save()
        userForm.save()

    # Até aqui, antes do context

    context = {
        'profileForm': profileForm,
        'userForm': userForm
    }

    return render(request, template_name='user/profile.html',
context=context, status=200)
```

Vamos entender o que mudamos em nosso código:

- **request.method**: informa o tipo de requisição que foi feita, `GET` ou `POST`.
- **request.POST**: dentro do request temos o `.POST`, com o qual podemos acessar qualquer dado enviado pelo formulário. Ele possui um dicionário do Django com todos os campos enviados pelo formulário. Usamos esse `request.POST` como parâmetro

para as classes de form `UserProfileForm` e `UserForm` informando para elas quais os novos valores que o formulário enviou para o servidor. Perceba que há uma verificação, `request.method == 'POST'` . Caso o tipo de método seja `GET` , significa que a página acabou de carregar pela primeira vez, então não devemos passar o valor de `request.POST` para as classes `UserProfileForm` e `UserForm` , mas apenas a instância que carregará e preencherá os valores atuais no formulário. Caso o método seja do tipo `POST` , entraremos no fluxo que passará como primeiro parâmetro para as classes `UserProfileForm` e `UserForm` o `request.POST` , para que possamos informar os novos valores para as duas classes de forms, assim futuramente salvaremos esses valores no lugar dos valores anteriores.

- **.is_valid()**: aqui, verificamos se o formulário recebeu valores válidos como parâmetro. No primeiro caso, quando passarmos `None` no `POST` , ele pulará esse fluxo e não chamará o método `save` , mas quando fizermos envios de dados pelo formulário, o método `is_valid()` retornará verdadeiro, caso todos os campos estejam corretos. Como `is_valid()` retornará True, ele permitirá que salvemos nossas models através do método `save()` que existe na classe do `django form` .
- **.save()**: método do `django form` que usamos para pegar os valores válidos do formulário e salvar na model que o formulário representa.

Perceba que o que faz esse formulário atualizar um item da model é o atributo `instance`. Como passamos uma instância para a classe do formulário junto com os novos valores, ele entende que os novos valores vão atualizar a instância. Se não passamos uma instância e somente passamos o `request.POST`, um novo item será criado no banco de dados, mas como na edição de perfil estamos somente atualizando, fazemos assim. Já em um formulário de cadastro de novo usuário, por exemplo, devemos passar o form sem a instância, algo como `ClasseDoFormulario(request.POST)`, e ele salvará como um novo dado. Aqui, `ClasseDoFormulario` representa uma suposição apenas, ela não existe em nosso sistema.

Mensagens de validação no formulário

Podemos também adicionar os campos de mensagem de validação no formulário de modo que ele informe ao usuário quando um campo está com formato inválido e não respeita as regras necessárias. Para isso, usamos o objeto `.errors` dentro de cada item do formulário, que retorna uma lista onde pode haver um ou mais erros de validação. Veja o código a seguir do arquivo `profile.html` da pasta `user`:

medicSearch/templates/user/profile.html

```
... código oculto
<form class="col-md-6 col-lg-6 offset-md-3 offset-lg-3" method="POST"
action="">
    <div class="row">
        {% csrf_token %}
        {% for f in userForm %}
            {% if not f.is_hidden %}
            <div class="form-group col-md-6">
                {{ f.label }}
                {{ f }}
                <!-- Adicione a linha a seguir -->
```

```
                {% for error in f.errors %}
                <div class="invalid-feedback" style="display: initial;">{{
error }}</div>
                {% endfor %}
                <!-- Até aqui -->
            </div>
            {% else %}
                {{ f }}
            {% endif %}
        {% endfor %}
        {% for f in profileForm %}
            {% if not f.is_hidden %}
            <div class="form-group col-md-6">
                {{ f.label }}
                {{ f }}
                <!-- Adicione a linha a seguir -->
                {% for error in f.errors %}
                <div class="invalid-feedback" style="display: initial;">{{
error }}</div>
                {% endfor %}
                <!-- Até aqui -->
            </div>
            {% else %}
                {{ f }}
            {% endif %}
        {% endfor %}
    </div>
    <button type="submit" class="btn btn-primary">Salvar</button>
</form>
... código oculto
```

Perceba que pegamos cada item `f`, que é uma instância de um `input` de formulário, e dentro dessa instância temos uma lista chamada `errors`. Com isso, criamos um `for` para iterar sobre os erros e, caso essa lista não esteja vazia, ou seja, caso haja algum erro no formulário, poderemos exibir no `HTML`:

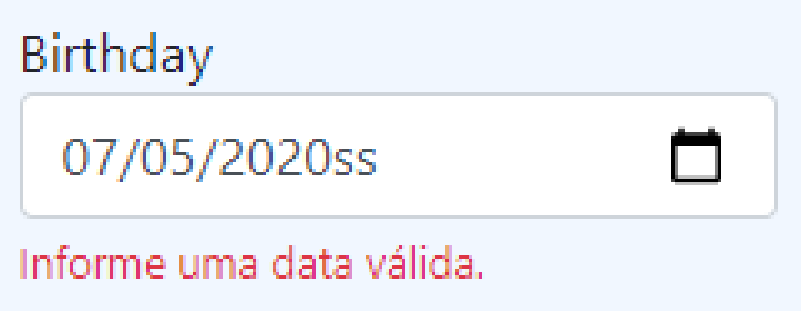


Figura 9.3: Formulário validação

Upload de imagem

Se você tentar enviar uma foto agora, provavelmente não conseguirá, porque precisamos adicionar um item à nossa def `edit_profile` . Abra o arquivo `ProfileView.py` dentro da pasta `medicSearch/views` e altere como no exemplo a seguir:

medicSearch/views/ProfileView.py

```
from django.core.paginator import Paginator
from django.shortcuts import render, redirect, get_object_or_404
from medicSearch.models import Profile
from medicSearch.forms.UserProfileForm import UserProfileForm, UserForm

def list_profile_view(request, id=None):
    # Código oculto

def edit_profile(request):
    profile = get_object_or_404(Profile, user=request.user)
    emailUnused = True

    if request.method == 'POST':
        # Altere a linha a seguir e adicione como segundo parâmetro da
classe `profileForm` o request.FILES
        profileForm = UserProfileForm(request.POST, request.FILES,
instance=profile)
        # Até aqui
        userForm = UserForm(request.POST, instance=request.user)
```

```
        verifyEmail =
Profile.objects.filter(user__email=request.POST['email']).exclude(user__id
=request.user.id).first()
        emailUnused = verifyEmail is None
    else:
        profileForm = UserProfileForm(instance=profile)
        userForm = UserForm(instance=request.user)

    if profileForm.is_valid() and userForm.is_valid():
        profileForm.save()
        userForm.save()

    context = {
        'profileForm': profileForm,
        'userForm': userForm
    }

    return render(request, template_name='user/profile.html',
context=context, status=200)
```

Ao adicionar o `request.FILES` como parâmetro da classe `UserProfileForm`, estamos dizendo ao Django que queremos salvar arquivos enviados via formulário.

Agora precisamos adicionar um item ao formulário para permitir que façamos upload de imagens no form. Altere o arquivo `profile.html` da pasta `templates/user` conforme o exemplo a seguir:

medicSearch/templates/user/profile.html

```
... código oculto
            <!-- Altere a linha a seguir adicionando
enctype="multipart/form-data" como atributo do form -->
            <form class="col-md-6 col-lg-6 offset-md-3 offset-lg-3"
method="POST" action="" enctype="multipart/form-data">
                ... código oculto não apague
            </form>
... código oculto
```

O `enctype="multipart/form-data"` é o responsável por dizer ao formulário que enviaremos arquivos em nossa requisição para o servidor. Com isso, já conseguimos salvar imagens em nosso banco de dados!

Lembre-se de que se for fazer upload de arquivos é necessário realizar esse processo, mas se não for fazer, não há necessidade de configurar sua classe de form para receber um `request.FILES` .

Mensagem de dados salvos

Para fechar nosso `ModelForm` , vamos adicionar uma mensagem de "dados salvos" ou de "erro ao salvar". Para isso, vamos abrir o arquivo `ProfileView.py` na pasta `medicSearch/views` e adicionar algumas coisas:

medicSearch/views/ProfileView.py

```
def edit_profile(request):
    profile = get_object_or_404(Profile, user=request.user)
    emailUnused = True

    # Adicione a linha a seguir
    message = None

    if request.method == 'POST':
        profileForm = UserProfileForm(request.POST, request.FILES,
instance=profile)
        userForm = UserForm(request.POST, instance=request.user)

        verifyEmail =
Profile.objects.filter(user__email=request.POST['email']).exclude(user__id
=request.user.id).first()
        emailUnused = verifyEmail is None
    else:
        profileForm = UserProfileForm(instance=profile)
        userForm = UserForm(instance=request.user)

    if profileForm.is_valid() and userForm.is_valid() and emailUnused:
```

```
        profileForm.save()
        userForm.save()
        # Adicione a linha a seguir
        message = { 'type': 'success', 'text': 'Dados atualizados com
sucesso' }
    # Adicione as linhas a seguir
    else:
        # Aqui verificamos se é do tipo post, para que na primeira vez que
a página carregar a mensagem não apareça, já que no primeiro carregamento
não enviamos um post, o form é dado como inválido e entra aqui.
        if request.method == 'POST':
            if emailUnused:
                # Se o e-mail não está em uso mas o formulário tiver algum
dado inválido.
                message = { 'type': 'danger', 'text': 'Dados inválidos' }
            else:
                # Se o e-mail que o usuário tentou usar já está em uso por
outra pessoa.
                message = { 'type': 'warning', 'text': 'E-mail já usado
por outro usuário' }

    # Adicione a chave message a seguir
    context = {
        'profileForm': profileForm,
        'userForm': userForm,
        'message': message
    }

    return render(request, template_name='user/profile.html',
context=context, status=200)
```

Com isso, vamos exibir uma mensagem de sucesso caso o formulário seja válido; caso estejamos carregando a tela pela primeira vez, `message` será do tipo `None` e não exibirá nada; e caso tentemos submeter o formulário e haja algum dado inválido, ele informará que os dados enviados são inválidos ou que o usuário está tentando salvar um e-mail que já existe na plataforma.

Agora precisamos adicionar um pequeno trecho de código ao html. Abra o arquivo `profile.html` dentro da pasta `user` e altere como no exemplo a seguir:

medicSearch/templates/user/profile.html

```
... código oculto
            <form class="col-md-6 col-lg-6 offset-md-3 offset-lg-3"
method="POST" action="" enctype="multipart/form-data">
                <!-- Adicione as linhas a seguir -->
                {% if message is not None %}
                <div class="alert alert-{{ message.type }}">{{
message.text }}</div>
                {% endif %}
                <!-- Até aqui -->
                <div class="row">
                    {% csrf_token %}
                    {% for f in userForm %}
                ... código oculto não apague
            </form>
... código oculto
```

Caso o dicionário `message` não seja `None` ele exibirá uma mensagem. Nossa tela ficará parecida com a imagem a seguir, acesse pela url `http://localhost:8000/profile/edit` :

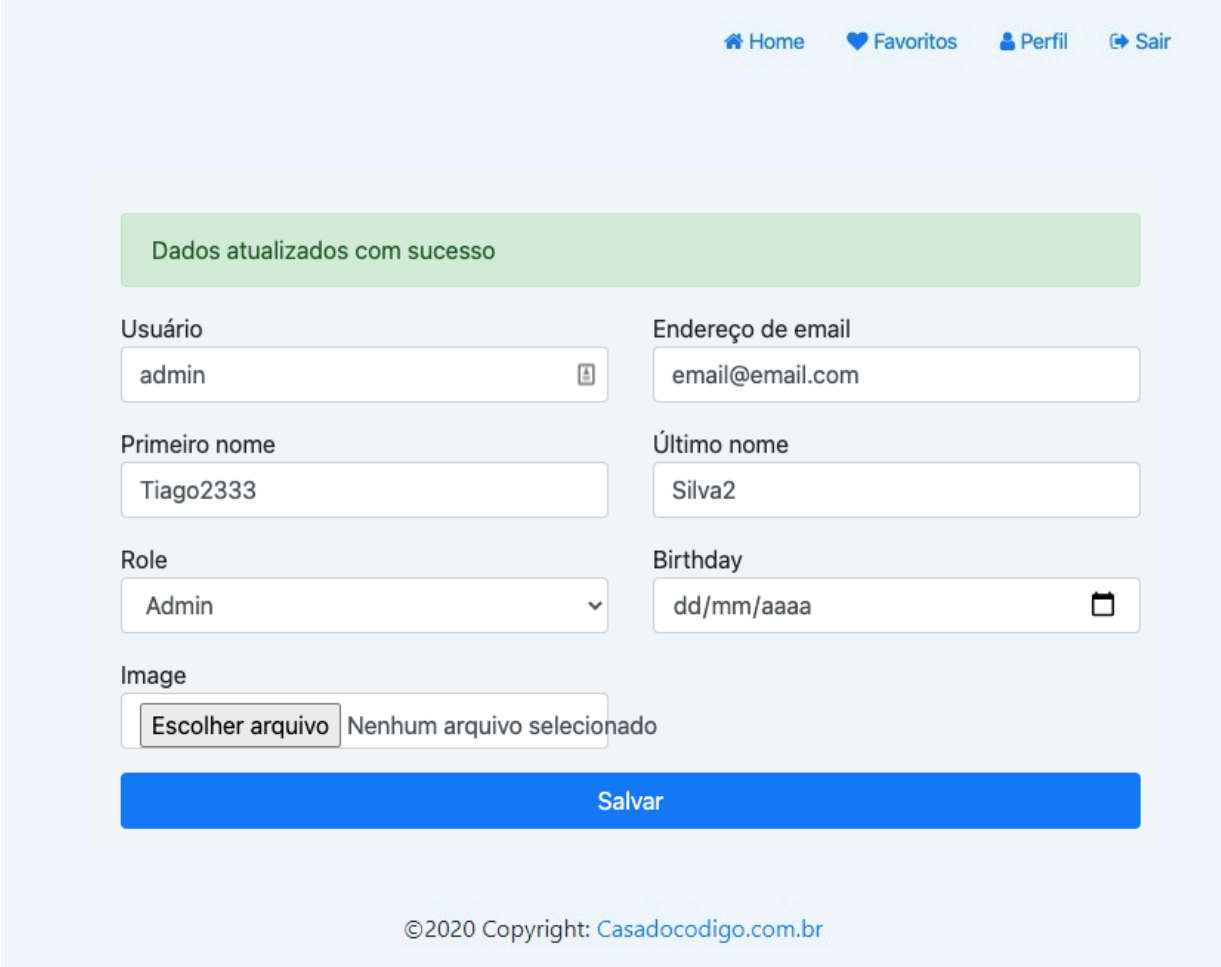


Figura 9.4: Formulário perfil - Mensagem de sucesso

No próximo capítulo, continuaremos a falar sobre formulários. Nele, aprenderemos a criar formulários customizados e criaremos as telas de cadastro e avaliação do médico.

Capítulo 10
Trabalhando com forms - Parte II

Dando continuidade ao último capítulo, vamos começar criando um formulário customizado para nossa aplicação. Ele será aplicado como nosso login do sistema.

10.1 Criando formulários customizados

Nessa etapa, aprenderemos a criar um form sem que ele espelhe a nossa model. Teremos a criação do formulário de cadastro no sistema e o de login. Nosso cadastro não terá confirmação por e-mail para não estender muito, mas aprenderemos a fazer recuperação de senha no capítulo sobre serviço de envio de e-mail, então você pode aplicar a mesma técnica para criar uma confirmação de cadastro por e-mail.

Tela de login

Vamos criar um formulário de login e também customizá-lo para que seja em uma url diferente do padrão. Para começar, dentro da pasta `medicSearch/forms` , crie um arquivo chamado `AuthForm.py` . Vamos criar nossa classe:

medicSearch/forms/AuthForm.py

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField(required=True,
widget=forms.TextInput(attrs={'class': 'form-control'}))
    password = forms.CharField(max_length=32,
```

```
widget=forms.PasswordInput(attrs={'class': 'form-control'}),
required=True)
```

Aqui não vemos muita coisa diferente da `ModelForm`. Podemos ver que, nesse modelo, nossa classe não herda da classe `ModelForm`, mas sim da classe `Form`, então não é necessário criarmos a classe `Meta` para configurar os campos que queremos exibir, já que não há uma model de referência para o formulário.

Devemos criar os atributos das classes que representarão cada campo que existirá no formulário. O módulo `form` possui várias classes dentro de si que são parecidas com as que usamos quando definimos um tipo de dado para um atributo em nossa model. A diferença é que aqui usamos o módulo `form` para selecionar o tipo de dado que nosso campo do formulário vai representar, já que essa classe não tem uma model como referência.

Perceba que passamos o atributo, o tipo do campo e dentro dele também podemos passar um `widget` que recebe uma classe que representa o tipo do input desse campo. Geralmente é algo como `TextInput`, `EmailInput`, `PasswordInput` etc. Vejamos alguns tipos de campos e seus respectivos tipos de widgets.

BooleanField

- Widget padrão: `CheckboxInput` (Campo de input checkbox)
- Valor vazio: falso
- Tipo de valor: um True ou False

CharField

- Widget padrão: `TextInput` (Campo de input text)
- Possui três argumentos opcionais:
  - `min_length` e `max_length`: se fornecidos, esses argumentos garantem que a sequência tenha no mínimo, ou no máximo, o comprimento especificado.
  - `strip`: se o valor for True, serão retirados os espaços em branco à esquerda e à direita do valor. True é o padrão

quando nada é passado.
    - `empty_value` : o valor a ser usado para representar "vazio". Podemos usar para passar um valor quando nada é colocado.

ChoiceFiled

- Widget padrão: `Select` (Campo de select html)
- Possui um argumento opcional:
    - `choices` : pode ser passada uma lista com tuplas de duas posições, ou um resultado vindo do banco que tenha esse formato. Padrão da tupla: `[('BR', 'Brasil')('EUA', 'Estados Unidos da América')]`.

MultipleChoiceField

- Widget padrão: `SelectMultiple` (Campo de select multiplo do html)
- Possui um argumento opcional:
    - `choices` : pode ser passada uma lista com tuplas de duas posições, ou um resultado vindo do banco que tenha esse formato. Padrão da tupla: `[('BR', 'Brasil')('EUA', 'Estados Unidos da América')]`.

DateField

- Widget padrão: `DateInput` (Campo de input text com padrão de data. Mesmo sendo campo de texto, o formato de data tem que ser respeitado para ser dado como campo válido).
- Possui um argumento opcional:
    - `input_formats` : uma lista de formatos usados para tentar converter uma sequência de caracteres em um objeto `datetime.date` válido. Exemplo: `forms.DateField(input_formats=['%d-%m-%Y'])`.

DateTimeField

- Widget padrão: `DateTimeInput` (Campo de input text com padrão de data. Mesmo sendo campo de texto o formato de data tem que ser respeitado para ser dado como campo válido).
- Possui um argumento opcional:
  - `input_formats` : uma lista de formatos usados para tentar converter uma sequência de caracteres em um objeto `datetime.datetime` válido. Exemplo: `forms.DateTimeField(input_formats=['%Y-%m-%d %H:%M']) .`

TimeField

- Widget padrão: `TimeInput` (Input text)
- Possui um argumento opcional:
  - `input_formats` : uma lista de formatos usados para tentar converter uma sequência de caracteres em um objeto `datetime.time` válido. Exemplo: `forms.TimeField(widget=forms.TimeInput(format='%H:%M')) .`

DurationField

- Widget padrão: `TextInput` (Campo de input Text).

FloatField

- Widget padrão: `NumberInput`
- Possui dois argumentos opcionais:
  - `min_length` e `max_length` : eles controlam a faixa de valores permitidos no campo.

DecimalField

- Widget padrão: `NumberInput`
- Possui quatro argumentos opcionais:
  - `min_value` e `max_value` : controlam o intervalo de valores permitido no campo e devem ser dados como decimais.
  - `max_digits` : o número máximo de dígitos (aqueles antes da vírgula decimal mais aqueles após a vírgula decimal, com zeros à esquerda removidos) permitido no valor.

    - `decimal_places` : o número máximo de casas decimais permitido.

IntegerField

- Widget padrão: `NumberInput`
- Possui dois argumentos opcionais:
    - `min_length` e `max_length` : controlam a faixa de valores permitidos no campo.

EmailField

- Widget padrão: `EmailInput` (Campo de input Email)
- Possui dois argumentos opcionais:
    - `min_length` e `max_length` : se fornecidos, esses argumentos garantem que a sequência tenha no máximo ou pelo menos o comprimento especificado.

FileField

- Widget padrão: `ClearableFileInput` (Input file com opção de limpar imagem)
- Possui dois argumentos opcionais:
    - `max_length` : se fornecido, assegura que o nome do arquivo tenha no máximo o comprimento especificado.
    - `allow_empty_file` : se fornecido, assegura que a validação seja bem-sucedida, mesmo que o conteúdo do arquivo esteja vazio.

ImageField

- Widget padrão: `ClearableFileInput` (Input file com opção de limpar imagem)
- Observação: o uso de um `ImageField` requer que o **Pillow** seja instalado com suporte para os formatos de imagem usados. Se você encontrar um erro de imagem corrompida ao fazer upload de uma imagem, geralmente significa que o Pillow não entende

seu formato. Para corrigir isso, instale a biblioteca apropriada e reinstale o Pillow.

Com isso, temos os tipos de campos mais usados na construção de um formulário customizado no Django.

Agora precisamos criar a `view` que realizará nosso login. Nessa etapa não há muita coisa diferente, todos os métodos da classe `form` do Django funcionam em forms customizados. O único que é uma exceção e não funcionará aqui é o método `.save()`, afinal, já que não há uma model como referência nesse formulário, ele não sabe onde deve salvar nada. Desse modo, em casos com formulários customizados, usamos o método `is_valid()` para validar e depois passamos os valores do `request` para a model ou as models que desejamos usar para salvar ou atualizar um item.

Neste momento, criaremos a lógica de login da aplicação em nossa `view`, sabendo que receberemos dois valores: o de username e o de password. Vamos lá, crie um arquivo na pasta `medicSearch/views` chamado `AuthView.py` e vamos adicionar o código a seguir.

medicSearch/views/AuthView.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login
from medicSearch.forms.AuthForm import LoginForm

def login_view(request):
    loginForm = LoginForm()
    message = None

    if request.user.is_authenticated:
        return redirect('/')

    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        loginForm = LoginForm(request.POST)
```

```
        if loginForm.is_valid():
            user = authenticate(username=username, password=password)
            if user is not None:
                login(request, user)
                return redirect('/')
            else:
                message = {
                    'type': 'danger',
                    'text': 'Dados de usuário incorretos'
                }

    context = {
        'form': loginForm,
        'message': message,
        'title': 'Login',
        'button_text': 'Entrar',
        'link_text': 'Registrar',
        'link_href': '/register'
    }


    return render(request, template_name='auth/auth.html',
context=context, status=200)
```

Grande parte é similar ao do modelo `ModelForm`, mas o que vemos de diferente nessa `view` é o método `authenticate()`, que recebe os parâmetros de `username` e `password`. Caso o login esteja correto, ele retornará o objeto do usuário `user` e entrará no fluxo de verificação `if user is not None`. Como `user` está correto e não retornou `None`, ele entrará nesse fluxo e chamará o método `login`, que recebe como parâmetro o `request` e o objeto `user`; caso o método `authenticate()` retorne `None`, significa que o usuário ou senha estão incorretos e assim será emitida uma mensagem na tela.

Perceba que criamos também quatro chaves de texto, `title`, `button_text`, `link_text` e `link_href`. Usaremos esses itens para dinamizar nosso arquivo `html`, pois desse modo também poderemos usar o template `html` de login para a tela de registro. Perceba que por isso chamamos o `html` de `auth.html` e não `login.html`, pois usaremos o mesmo template para `login` e para registrar novos usuários. Esses campos serão mudados quando criarmos a `view` de registro que apontará para o mesmo template. Mais à frente entenderemos melhor.

Com nosso arquivo criado, vamos adicioná-lo ao arquivo `__init__.py`, que fica na mesma pasta `views`:

medicSearch/views/__init__.py

```
from .HomeView import *
from .ProfileView import *
from .MedicView import *
# Adicione a linha a seguir
from .AuthView import *
```

Agora, vamos criar a rota que será responsável pela tela de login. Dentro da pasta `medicSearch/urls`, crie um arquivo chamado `AuthUrls.py` e adicione o código a seguir:

medicSearch/urls/AuthUrls.py

```
from django.urls import path
from medicSearch.views.AuthView import login_view

urlpatterns = [
    path("login", login_view, name='login'),
]
```

Com nosso arquivo criado, vamos adicioná-lo ao arquivo `__init__.py`, que fica na mesma pasta `urls`:

medicSearch/urls/__init__.py

```
from .HomeUrls import *
from .ProfileUrls import *
from .MedicUrls import *
# Adicione a linha a seguir
from .AuthUrls import *
```

Precisamos registrar nosso arquivo de url que criamos chamado `AuthUrls.py` no arquivo de urls principal, que fica na pasta `medicSearchAdmin`, no arquivo `urls.py`. Abra esse arquivo e altere-o como no exemplo:

medicSearchAdmin/urls.py

```
from django.contrib import admin
from django.urls import path
from django.conf.urls import url, include
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('medicSearch.urls.HomeUrls')),
    # Adicione a linha a seguir
    path('', include('medicSearch.urls.AuthUrls')),
    # Até aqui
    path('profile/', include('medicSearch.urls.ProfileUrls')),
    path('medic/', include('medicSearch.urls.MedicUrls')),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT) +
static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Agora temos nossa url funcionando e precisamos criar um arquivo html para ser nosso template. No arquivo `AuthView.py`, nós já deixamos a `view` apontando para um template chamado `auth/auth.html`, mas ele ainda não existe. Então, dentro da pasta `templates`, crie uma pasta chamada `auth` e, dentro dela, um arquivo chamado `auth.html` e adicione o código base a seguir:

medicSearch/templates/auth/auth.html

```
{% extends "base.html" %}
{% load static %}
{% block title %}{{ title }}{% endblock %}
{% block styles %}<link rel="stylesheet" href="{% static 'css/home.css'
%}">{% endblock %}
{% block content %}
<div id="content">
    <div class="container-fluid">
        <div class="row">
            <form class="col-md-4 col-lg-4 offset-md-4 offset-lg-4"
method="POST" action="">
                <h3 class="text-center">{{title}}</h3>
                {% if message is not None %}
                <div class="alert alert-{{ message.type }}">{{
message.text }}</div>
                {% endif %}
                {% csrf_token %}
                {% for f in form %}
                    <div class="form-group">
                        {{ f.label }}
                        {{ f }}
                        {% for error in f.errors %}
                        <div class="invalid-feedback" style="display:
initial;">{{ error }}</div>
                        {% endfor %}
                    </div>
                {% endfor %}
                <a href="{{link_href}}" style="
                    display: block;
                    text-align: center;
                    margin-bottom: 20px;
                ">{{link_text}}</a>
                <button type="submit" class="btn btn-primary">
{{button_text}}</button>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Como já sabemos o que cada parte do form faz, não é necessário explicar nada nessa etapa. Já podemos testar nosso login e ele retornará a mensagem de erro caso tentemos entrar com um usuário incorreto. Veja acessando a url `http://localhost:8000/login`.

> Como ainda não criamos o link de logout, vá até `http://localhost:8000/admin` para realizar o logout, depois vá para o `/login`. A tela de login à qual o admin vai redirecionar não é a mesma que criamos agora. A seguir resolveremos isso também, por enquanto após deslogar vá para a url manualmente.

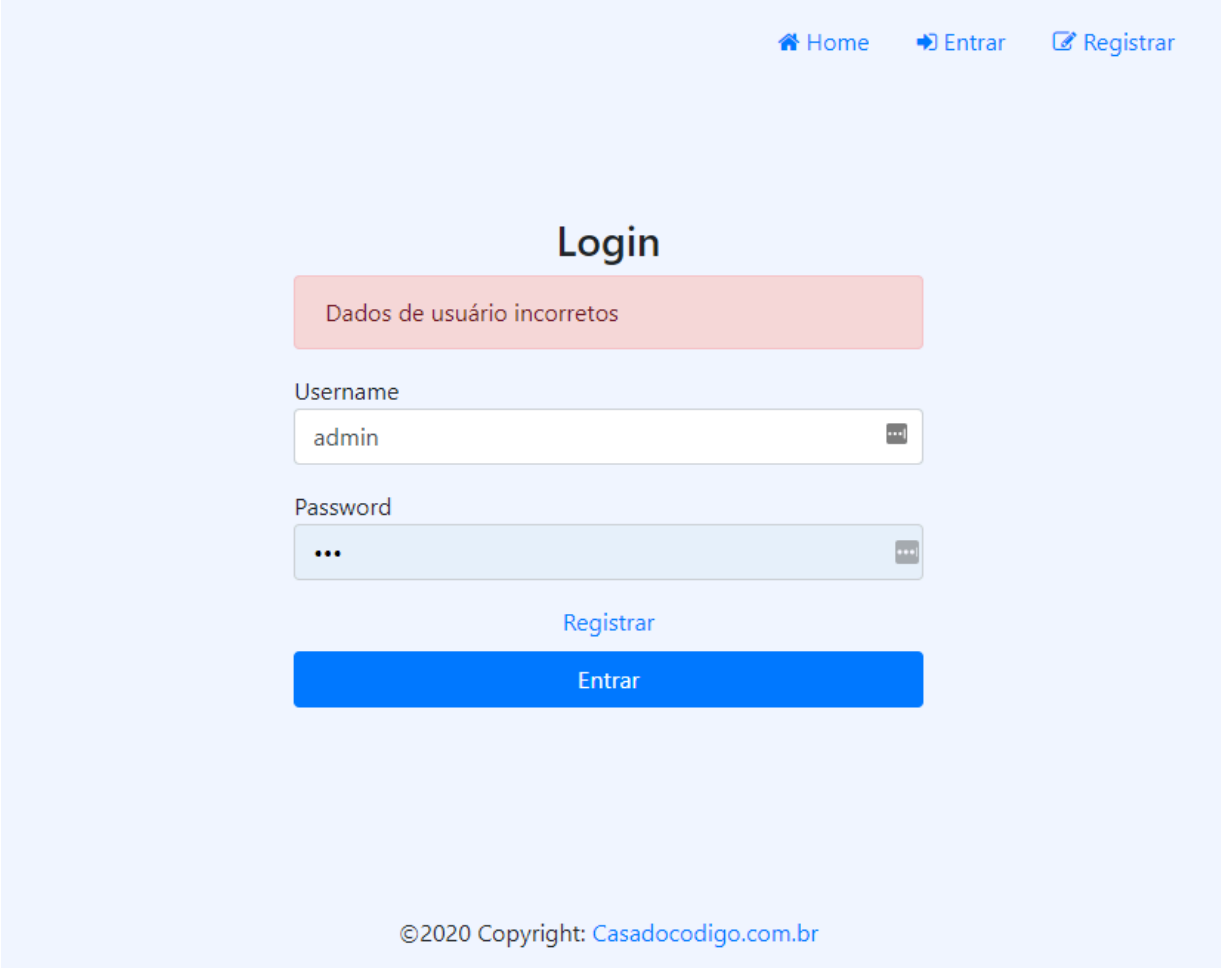


Figura 10.1: Formulário login

Se tentarmos realizar o login por essa página, já conseguiremos entrar também.

Uma configuração que eu recomendo é adicionar quatro atributos básicos ao arquivo de `settings.py`, que são necessários para o fluxo de `login` e `logout` funcionarem corretamente (faremos o logout mais à frente). Abra o arquivo `settings.py` dentro da pasta `medicSearchAdmin\settings` e adicione as linhas a seguir no final do arquivo:

medicSearchAdmin/settings/settings.py

```
# Adicione ao final do arquivo
LOGIN_URL = '/login'
LOGIN_REDIRECT_URL = '/'
LOGOUT_URL = '/logout'
LOGOUT_REDIRECT_URL = '/login'
```

Vamos entender cada atributo:

- **LOGIN_URL**: define a rota padrão de login do sistema.
- **LOGIN_REDIRECT_URL**: define para onde seremos redirecionados caso o login ocorra com sucesso.
- **LOGOUT_URL**: define a rota padrão de logout do sistema.
- **LOGOUT_REDIRECT_URL**: define para onde seremos redirecionados caso o logout ocorra com sucesso.

Qualquer tipo de usuário poderá fazer login por essa tela. Se o usuário for um administrador, poderá acessar o painel administrativo pela url `/admin` da mesma maneira como era feito antigamente.

Agora vamos criar nossa tela de cadastro, que permitirá que um usuário se cadastre em nossa plataforma.

10.2 Tela de cadastro

Essa tela não será muito diferente da tela de login, mas conterá o campo de e-mail que futuramente usaremos para confirmação de e-mail que aprenderemos a fazer alguns capítulos a seguir.

Vamos começar criando a classe `RegisterForm` dentro do arquivo `AuthForm.py`, que fica na pasta `medicSearch/forms`. Veja o exemplo a seguir:

medicSearch/forms/AuthForm.py

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField(required=True,
widget=forms.TextInput(attrs={'class': 'form-control'}))
    password = forms.CharField(max_length=32,
widget=forms.PasswordInput(attrs={'class': 'form-control'}),
required=True)

# Adicione a classe a seguir
class RegisterForm(forms.Form):
    username = forms.CharField(required=True,
widget=forms.TextInput(attrs={'class': 'form-control'}))
    email = forms.CharField(required=True, widget=forms.EmailInput(attrs=
{'class': 'form-control'}))
    password = forms.CharField(max_length=32,
widget=forms.PasswordInput(attrs={'class': 'form-control'}),
required=True)
```

Perceba que, no campo de e-mail, alteramos nosso `widget` para ser um input de tipo e-mail. Isso será útil para que a validação do campo exija que um e-mail seja adicionado a ele.

Agora precisamos criar a view de registro. Diferente do login, essa view usará o método `.save()` da classe `form`, porém não passaremos uma instância inicial para o método `form`. Desse modo,

o `.save()` criará um novo usuário em vez de editar um já existente no sistema. Vamos lá.

Abra o arquivo `AuthView.py` dentro da pasta `medicSearch/views` e vamos criar a `view` de registro:

medicSearch/views/AuthView.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login
# Adicione as linhas a seguir
from django.contrib.auth.models import User
# Adicione no import a seguir a classe RegisterForm
from medicSearch.forms.AuthForm import LoginForm, RegisterForm

def login_view(request):
    ...código oculto para simplificar. Não apague.
# Adicione a classe a seguir
def register_view(request):
    registerForm = RegisterForm()
    message = None

    if request.user.is_authenticated:
        return redirect('/')

    if request.method == 'POST':
        username = request.POST['username']
        email = request.POST['email']
        password = request.POST['password']
        registerForm = RegisterForm(request.POST)

        if registerForm.is_valid():
            # Aqui verificamos se existe usuário ou e-mail com esse
cadastro
            verifyUsername =
User.objects.filter(username=username).first()
            verifyEmail = User.objects.filter(email=email).first()

            if verifyUsername is not None:
                message = { 'type': 'danger', 'text': 'Já existe um
```

```
usuário com este username!' }
            elif verifyEmail is not None:
                message = { 'type': 'danger', 'text': 'Já existe um
usuário com este e-mail!' }
            else:
                user = User.objects.create_user(username, email, password)
                if user is not None:
                    message = { 'type': 'success', 'text': 'Conta criada
com sucesso!' }
                else:
                    message = { 'type': 'danger',  'text': 'Um erro
ocorreu ao tentar criar o usuário.' }

    context = {
        'form': registerForm,
        'message': message,
        'title': 'Registrar',
        'button_text': 'Registrar',
        'link_text': 'Login',
        'link_href': '/login'
    }
    return render(request, template_name='auth/auth.html',
context=context, status=200)
```

Podemos perceber que surgiu mais um método nessa view, o `create_user`. Poderíamos ter criado o usuário manualmente, mas usar esse método traz algumas vantagens, como a validação do usuário para saber se ele já existe, ou se o username ou password não atendem às expectativas, por exemplo.

Perceba também que usaremos o mesmo template `html`, já que a aparência do formulário e as mensagens de erro e sucesso serão iguais. Veja que no dicionário de `context` agora passamos para `form` a instância `registerForm`, que representa a instância da classe `RegisterForm()`.

Desse modo, em login, a chave `form` do `context` representa a instância da classe `LoginForm()` e na `register`, representa a da

`RegisterForm()`. Assim conseguimos ter menos retrabalho na manutenção do nosso código `html`.

Agora precisamos adicionar essa view em nosso arquivo de url de autenticação. Para isso, vamos abrir o arquivo `AuthUrls.py` que fica dentro da pasta `medicSearch/urls` e adicionar as alterações a seguir:

medicSearch/urls/AuthUrls.py

```
from django.urls import path
# Adicione na linha a seguir o método register_view
from medicSearch.views.AuthView import login_view, register_view

urlpatterns = [
    path("login", login_view, name='login'),
    # Adicione a linha a seguir
    path("register", register_view, name='register'),
]
```

Com isso, já temos nosso formulário de registro funcionando perfeitamente e, ao registrar, o usuário já poderá realizar o login na plataforma.

Teremos um registro parecido com este se acessarmos `http://localhost:8000/register`:

Erro

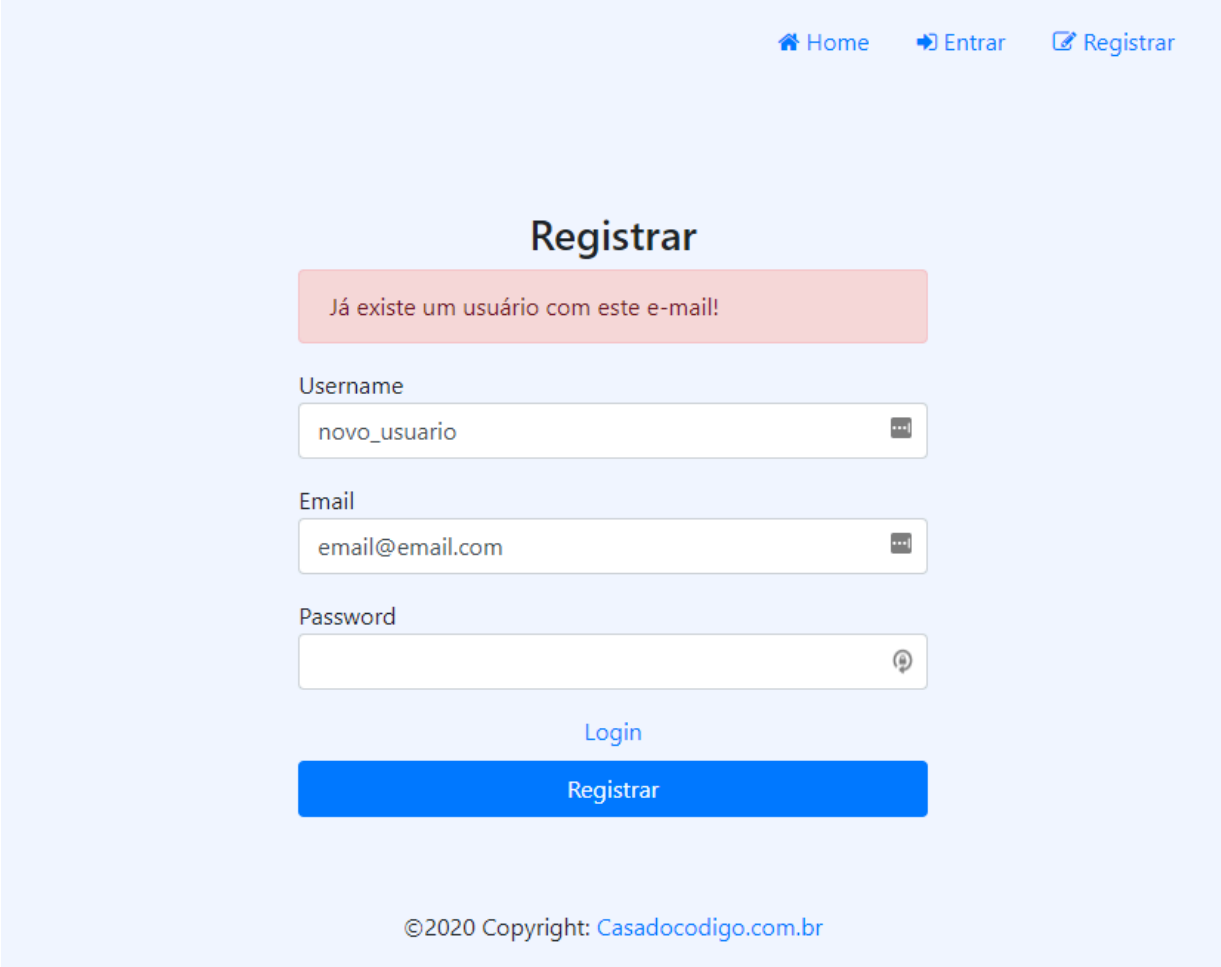


Figura 10.2: Formulário registro - Erro

Sucesso

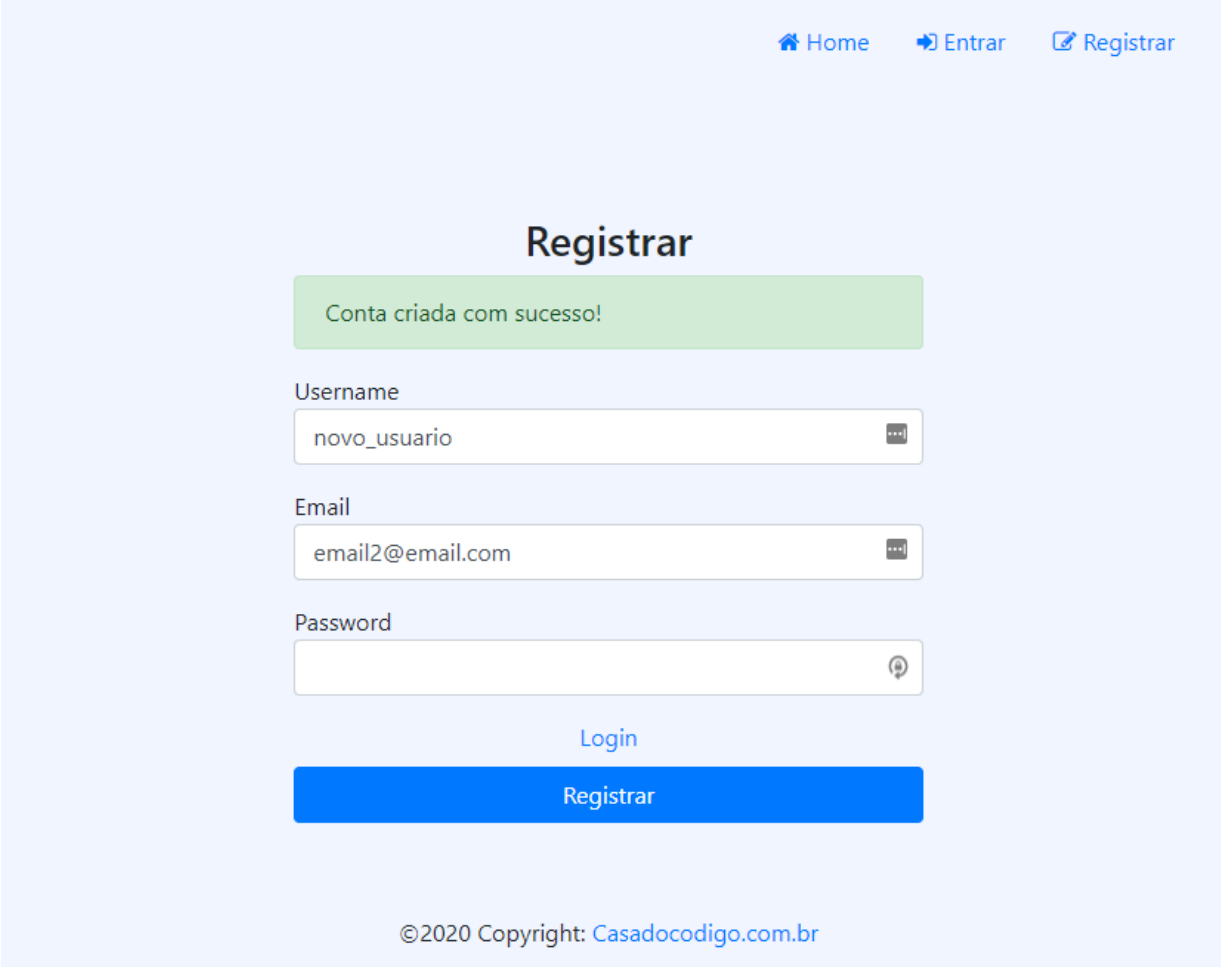


Figura 10.3: Formulário registro - Sucesso

Com isso, nosso fluxo de cadastro externo está completo. Para fechar, vamos criar rapidamente um formulário para avaliar o médico. Ele será interessante porque teremos uma regra de negócio que nos permitirá editar a avaliação, mas não poderemos criar mais de uma avaliação por médico, isto é, cada usuário poderá avaliar o médico uma única vez.

10.3 Avaliação do médico

Vamos começar criando uma classe `form` para a avaliação do médico. Crie um arquivo chamado `MedicForm.py` dentro da pasta

`medicSearch/forms` e vamos customizar nosso formulário:

medicSearch/forms/MedicForm.py

```
from django.forms import ModelForm
from django import forms
from medicSearch.models.Rating import Rating

class MedicRatingForm(ModelForm):
    class Meta:
        model = Rating
        fields = ['user', 'user_rated', 'value', 'opinion']
        widgets = {
            'user': forms.HiddenInput(),
            'user_rated': forms.HiddenInput(),
            'value': forms.NumberInput(attrs={'class': "form-control"}),
            'opinion': forms.Textarea(attrs={'class': "form-control",
"rows": 4}),
        }
```

Com isso, temos nosso formulário de avaliação de médico criado. Vamos agora criar um método para avaliação de médico dentro do arquivo view `MedicView.py` , que fica na pasta `medicSearch/views` . Abra o arquivo e adicione as modificações como no exemplo a seguir:

medicSearch/views/MedicView.py

```
from django.shortcuts import render, redirect
# Altere a linha a seguir para importar a model Rating
from medicSearch.models import Profile, Rating
# Adicione a linha a seguir para importar a classe de form MedicRatingForm
from medicSearch.forms.MedicForm import MedicRatingForm
from django.db.models import Q
from django.core.paginator import Paginator

def list_medics_view(request):
    ...código oculto não apague

Adicione o método a seguir no final do arquivo após a função
remove_favorite_view
```

```
def rate_medic(request, medic_id=None):
    medic = Profile.objects.filter(user__id=medic_id).first()
    rating = Rating.objects.filter(user=request.user,
user_rated=medic.user).first()
    message = None
    initial = {'user': request.user, 'user_rated': medic.user}

    if request.method == 'POST':
        ratingForm = MedicRatingForm(request.POST, instance=rating,
initial=initial)
    else:
        ratingForm = MedicRatingForm(instance=rating, initial=initial)

    if ratingForm.is_valid():
        ratingForm.save()
        message = {'type': 'success', 'text': 'Avaliação salva com
sucesso'}
    else:
        if request.method == 'POST':
            message = {'type': 'danger', 'text': 'Erro ao salvar
avaliação'}

    context = {
        'ratingForm': ratingForm,
        'medic': medic,
        'message': message
    }

    return render(request, template_name='medic/rating.html',
context=context, status=200)
```

Esse método é bem simples: caso tenha uma avaliação já criada, ela será carregada no formulário, permitindo sua edição; caso não haja, uma avaliação será criada. Tanto a criação como a atualização ocorrem no método `.save()` do mesmo jeito que fizemos na atualização de perfil. Vale lembrar que aqui temos um modelo de formulário do tipo `ModelForm` que se baseia em uma model, por isso podemos usar o método `.save()` da classe form.

Perceba que agora passamos um atributo chamado `initial` para nossa classe de formulário. Vamos usá-lo para preencher o formulário informando quem avaliou e quem está sendo avaliado. Como os campos `user` e `user_rated` foram modificados para `hidden`, o usuário não poderá vê-lo e nem alterá-lo, desse modo cabe a nós deixar os valores desses campos já preenchidos.

Agora precisamos criar a rota que levará o usuário para a tela do formulário de avaliação. Vamos abrir o arquivo `MedicUrls.py`, que fica na pasta `medicSearch/urls` e adicionar as alterações a seguir:

medicSearch/urls/MedicUrls.py

```
from django.urls import path
# Adicione na importação a seguir a função rate_medic
from medicSearch.views.MedicView import list_medics_view,
add_favorite_view, remove_favorite_view, rate_medic

urlpatterns = [
    path("", list_medics_view, name='medics'),
    path("favorite", add_favorite_view, name='medic-favorite'),
    path("favorite/remove", remove_favorite_view, name='medic-favorite-
remove'),
    # Adicione a linha a seguir para criar a url de avaliação de médico
    path("rating/<int:medic_id>", rate_medic, name='rate-medic')
]
```

Se repararmos, não existe uma url que nos leve para a avaliação do médico ainda. Antes de criar o template para o formulário de avaliação, vamos adicionar um botão que levará o usuário para a página de avaliação do médico.

Abra o arquivo `profile.html` que fica dentro da pasta `medicSearch/templates/profile` e vamos adicionar o botão como no exemplo a seguir:

medicSearch/templates/profile/profile.html

```
... código oculto para reduzir e facilitar a leitura não apague
        <ul class="list-group">
            <li class="list-group-item">Nome: {{profile.user.get_full_name
| default:"Sem nome"}}</li>
            <li class="list-group-item">Usuário: {{profile.user.username |
default:"Sem usuário"}}</li>
            <li class="list-group-item">E-mail: {{profile.user.email |
default:"Sem e-mail"}}</li>
            <li class="list-group-item">Nascimento: {{profile.birthday |
date:'d/m/Y' | default:"Sem data"}}</li>
            {% if profile.role == 2 %}
            <li class="list-group-item">Nota: {{
profile.show_scoring_average }}</li>
            <li class="list-group-item">Especialidades: {{
profile.specialties.all | join:", " }}</li>
            {% endif %}
            <!-- Adicione a linha a seguir no menu da página do perfil -->
            {% if profile.role == 2 and request.user.is_authenticated %}
            <li class="list-group-item">
                <a class="btn btn-warning" href={% url 'rate-medic'
profile.user.id %}>Avaliar <i class="fa fa-star"></i></a>
            </li>
            {% endif %}
            <!-- Até aqui -->
        </ul>
    {% if profile.role == 1 or profile.role == 3 %}
    ... código oculto para reduzir e facilitar a leitura não apague
... código oculto para reduzir e facilitar a leitura não apague
{% endblock %}
```

Agora temos um link que nos levará para a avaliação do médico. Esse link só vai aparecer quando o usuário estiver logado e apenas em páginas de perfil de médicos. Então precisamos criar nosso template `html`.

Vamos criar um arquivo chamado `rating.html` dentro da pasta `medicSearch/templates/medic` e adicionar o código `html` a seguir:

medicSearch/templates/medic/rating.html

```
{% extends "base.html" %}
{% load static %}
{% block title %}Perfil{% endblock %}
{% block styles %}<link rel="stylesheet" href="{% static 'css/home.css'
%}">{% endblock %}
{% block content %}
<div id="content">
    <div class="container-fluid">
        <div class="row">
            <form class="col-md-4 col-lg-4 offset-md-4 offset-lg-4"
method="POST" action="">
                <h3 class="text-center">Avaliação do médico {{
medic.user.get_full_name }}</h3>
                {% if message is not None %}
                <div class="alert alert-{{ message.type }}">{{
message.text }}</div>
                {% endif %}
                {% csrf_token %}
                {% for f in ratingForm %}
                    {% if not f.is_hidden %}
                    <div class="form-group">
                        {{ f.label }}
                        {{ f }}
                        {% for error in f.errors %}
                        <div class="invalid-feedback" style="display:
initial;">{{ error }}</div>
                        {% endfor %}
                    </div>
                    {% else %}
                        {{ f }}
                    {% endif %}
                {% endfor %}
                <a href={% url 'profile' medic.user.id %} class="text-
center mt-2 mb-2" style="display: block;">Voltar para o perfil do
médico</a>
                <button type="submit" class="btn btn-
primary">Avaliar</button>
            </form>
        </div>
```

```
    </div>
</div>
{% endblock %}
```

O código é bem parecido com o de edição do perfil, com algumas diferenças de lógica na `view`, mas no html ele é praticamente igual. E com isso, temos uma tela de avaliação do médico, para que o usuário logado possa fazer uma avaliação. A tela será algo similar à imagem a seguir, para ver, você precisa ir ao perfil de um médico e clicar no botão avaliar que fica no menu, a seguir você verá as duas telas em sequência:

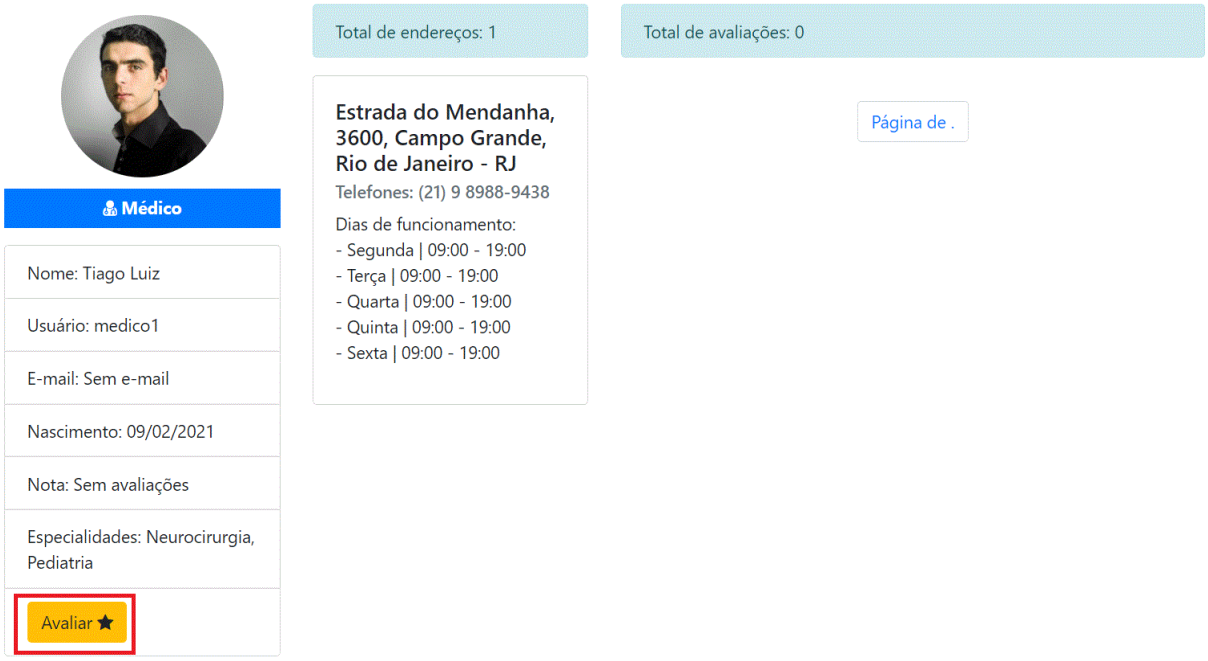


Figura 10.4: Formulário de avaliação - Perfil com botão

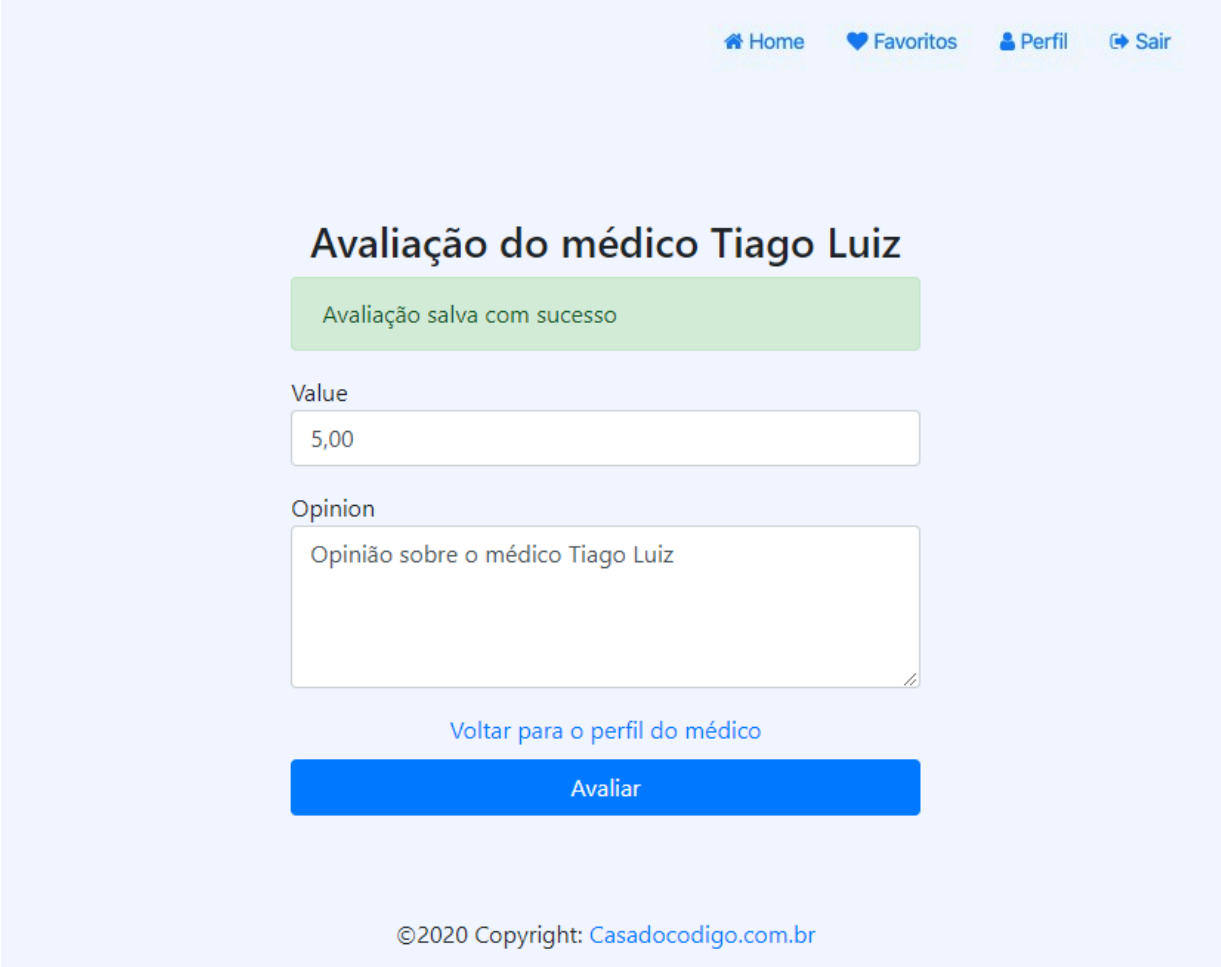


Figura 10.5: Formulário de avaliação - Sucesso

Existem algumas coisas que poderíamos fazer para melhorar nossa url de avaliação, como verificar se o id passado na url é evidentemente o de um médico ou o de algum outro usuário; caso não seja um médico, poderíamos redirecionar para a página do perfil do usuário novamente, informando que só médicos podem ser avaliados.

Muitas coisas podem ser feitas para melhorar o fluxo e a usabilidade do sistema. Fique à vontade para ampliar tudo que você aprendeu aqui trazendo soluções inovadoras e escaláveis.

Pronto, todos os formulários de registro do sistema estão criados, então agora estamos prontos para começar a controlar o fluxo de quem entra e sai da nossa plataforma. No próximo capítulo, aprenderemos a limitar o acesso a telas internas somente para quem estiver logado, e somente verá as telas de registro e login quem não estiver logado. Vamos nessa!

Capítulo 11
Trabalhando com autenticação

Neste capítulo, começaremos a criar os bloqueios do nosso sistema. Para acessar algumas páginas, como edição de perfil, é necessário realizar o login.

11.1 login_required

O Django possui um módulo que utilizamos para informar a uma `view` que o usuário precisa se autenticar para acessá-la. Vamos alterar nossas views que precisam de autenticação para serem acessadas. Abra o arquivo `MedicView.py` que fica na pasta `medicSearch/views` e adicione a alteração a seguir:

medicSearch/views/MedicView.py

```
# Adicione a linha a seguir
from django.contrib.auth.decorators import login_required
# Até aqui
from django.shortcuts import render, redirect
from medicSearch.models import Profile, Rating
from medicSearch.forms.MedicForm import MedicRatingForm
from django.db.models import Q
from django.core.paginator import Paginator

def list_medics_view(request):
    ...código oculto não apague nada em seu arquivo

# Este trecho está oculto mas possui outros métodos que não devem ser
apagados

# Adicione a linha a seguir
@login_required
```

```
def rate_medic(request, medic_id=None):
    ...código oculto não apague nada em seu arquivo
```

Vamos agora adicionar o `@login_required` no arquivo `ProfileView.py` que fica na pasta `medicSearch/views` :

medicSearch/views/ProfileView.py

```
from django.core.paginator import Paginator
# Adicione a linha a seguir
from django.contrib.auth.decorators import login_required
# Até aqui
from django.shortcuts import render, redirect, get_object_or_404
from medicSearch.models import Profile
from medicSearch.forms.UserProfileForm import UserProfileForm, UserForm

def list_profile_view(request, id=None):
    # ...código oculto não apague nada em seu arquivo

# Adicione a linha a seguir
@login_required
def edit_profile(request):
    # ...código oculto não apague nada em seu arquivo
```

Com isso, as duas views que precisam de login já estão com a exigência de login. Agora vamos usar o atributo `next` da url para direcionar o usuário para a página que ele queria após realizar login.

Usando o parâmetro next

Vamos abrir o arquivo `AuthView.py` que fica na pasta `medicSearch/views` e fazer algumas pequenas alterações para que ele redirecione o usuário para a tela que ele queria ir caso haja um parâmetro `next` na url:

medicSearch/views/AuthView.py

```
...código oculto não apague
        if loginForm.is_valid():
            user = authenticate(username=username, password=password)
```

```
            if user is not None:
                login(request, user)
                # Adicione as linhas a seguir no lugar do `return
redirect('/')`
                _next = request.GET.get('next')
                if _next is not None:
                    return redirect(_next)
                else:
                    return redirect("/")
                # Até aqui
            else:
                message = {
                    'type': 'danger',
                    'text': 'Dados de usuário incorretos'
                }
...código oculto não apague
```

Perceba que, agora, se houver um parâmetro chamado `next` na url, vamos redirecionar o usuário para a url deste parâmetro, ou seja, se a url for `http://localhost:8000/login?next=/medic/rating/1` , o usuário será redirecionado para a url `http://localhost:8000/medic/rating/1` automaticamente.

11.2 Urls do menu

Agora que nosso projeto está praticamente finalizado, já podemos criar nossa url de logout e também adicionar todas as urls correspondentes aos menus, que até agora não estão navegáveis. Vamos começar criando a `view` de logout no arquivo `AuthView.py` , que fica na pasta `medicSearch/views` . Adicione o código a seguir após os métodos `login_view` e `register_view` :

medicSearch/views/AuthView.py

```
from django.shortcuts import render, redirect
# Adicione o método `logout` na linha a seguir
```

```
from django.contrib.auth import authenticate, login, logout
# Até aqui
from django.contrib.auth.models import User
from medicSearch.forms.AuthForm import LoginForm, RegisterForm

def login_view(request):
    ...código oculto não remover nada

def register_view(request):
    ...código oculto não remover nada

def logout_view(request):
    logout(request)
    return redirect('/login')
```

Agora que nossa `view` está criada, vamos adicionar a url de logout ao arquivo `AuthUrls.py` que fica na pasta `medicSearch/urls` :

medicSearch/urls/AuthUrls.py

```
from django.urls import path
# Importe a função logout_view a seguir
from medicSearch.views.AuthView import login_view, register_view,
logout_view

urlpatterns = [
    path("login", login_view, name='login'),
    path("register", register_view, name='register'),
    # Adicione a linha a seguir
    path("logout", logout_view, name='logout'),
]
```

Pronto, agora só precisamos adicionar a url de logout ao nosso menu do template. Já que adicionaremos essa url, vamos aproveitar e configurar todas as urls do menu de uma vez só. Abra o arquivo `base.html` , que fica na pasta `medicSearch/templates` , e vamos alterá-lo:

medicSearch/templates/base.htm

```
...código oculto não apague
    <header>
        <nav class="navbar justify-content-end navbar-expand-lg">
            <ul class="nav">
                <li class="nav-item"><a class="nav-link" href="/"><i
class="fa fa-home"></i> Home</a></li>
                {% if user.is_authenticated %}
                    <li class="nav-item"><a class="nav-link" href={% url
'profiles' %}><i class="fa fa-user"></i> Perfil</a></li>
                    <li class="nav-item"><a class="nav-link" href={% url
'edit-profile' %}><i class="fa fa-edit"></i> Editar Perfil</a></li>
                    <li class="nav-item"><a class="nav-link" href={% url
'logout' %}><i class="fa fa-sign-out"></i> Sair</a></li>
                {% else %}
                    <li class="nav-item"><a class="nav-link" href={% url
'login' %}><i class="fa fa-sign-in"></i> Entrar</a></li>
                    <li class="nav-item"><a class="nav-link" href={% url
'register' %}><i class="fa fa-edit"></i> Registrar</a></li>
                {% endif %}
            </ul>
        </nav>
    </header>
...código oculto não apague
```

Perceba que adicionamos os caminhos aos atributos `href` para que nosso menu funcione corretamente. Com isso, nossa aplicação já ganhou forma. No próximo capítulo, aprenderemos a realizar login com redes sociais.

Conteúdo extra

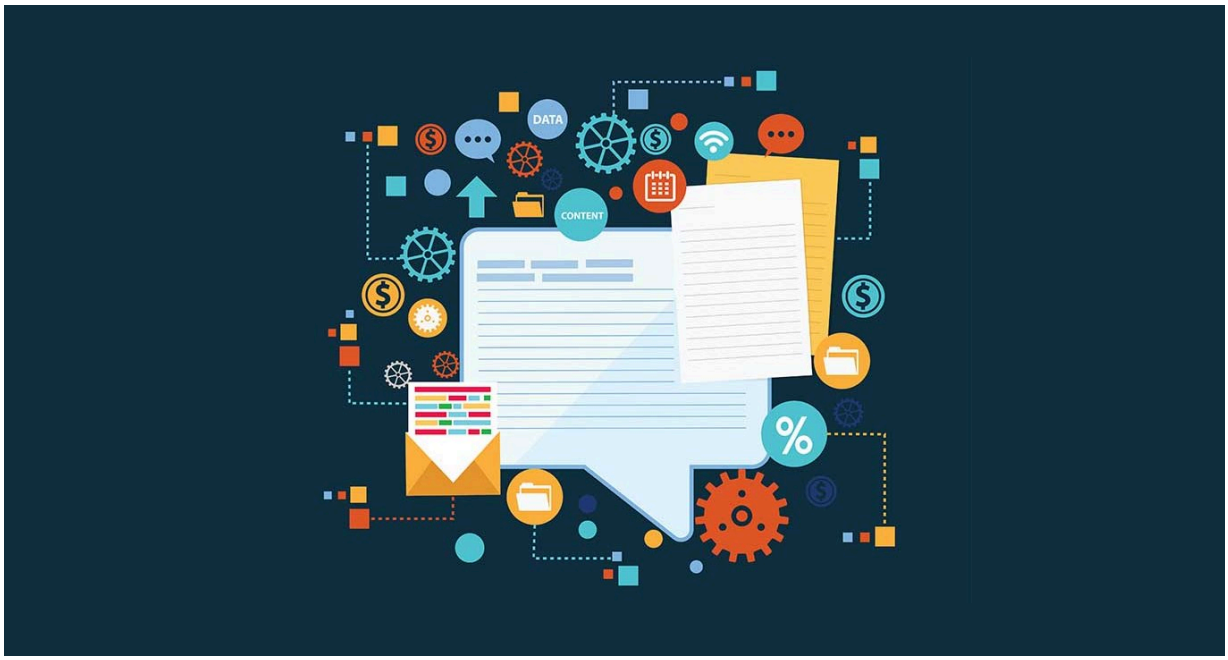


Figura 1: Conteúdo extra

Nesta unidade, veremos alguns conteúdos extras que serão muito importantes como complementos da sua aplicação Django, dentre os quais temos a autenticação com redes sociais e serviços de envio de e-mail.

Esses assuntos são diferenciais para a construção de uma boa aplicação feita em Django, mas lembre-se, sua aplicação já funciona corretamente com o que fizemos até agora. Com o que foi aprendido até o capítulo 11, você é inteiramente capaz de desenvolver um sistema totalmente profissional e que atenda às necessidades de seu `cliente/empresa/gestor`.

Capítulo 12
Autenticação com redes sociais

Nesta etapa, aprenderemos a adicionar o recurso de cadastro e login por redes sociais à nossa plataforma. Com isso, o usuário não precisa obrigatoriamente se cadastrar na plataforma de forma convencional.

Neste capítulo, veremos como configurar o Facebook e o Google, que são os mais utilizados atualmente.

12.1 Instalação

Para começar, precisamos instalar um módulo chamado `social-auth-app-django`. Ele nos auxiliará na configuração da autenticação através das redes sociais. Vamos começar instalando-o no virtualenv do nosso projeto.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install social-auth-app-django
```

Após realizarmos a instalação do módulo, vamos injetar o `social_django` no `INSTALLED_APPS`, que fica no arquivo `settings.py` dentro da pasta `medicSearchAdmin/settings`. Como o módulo `social-auth-app-django` é uma biblioteca adicional que complementa a aplicação Django, precisamos adicioná-la aos `INSTALLED_APPS` para que nossa aplicação a reconheça e funcione corretamente. Vamos fazer isso agora:

medicSearchAdmin/settings/settings.py

```
...código oculto
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
```

```
    'medicSearch',
    # Adicione a linha a seguir no `INSTALLED_APPS`
    'social_django'
]

...código oculto

# Adicione as linhas a seguir no final do arquivo
AUTHENTICATION_BACKENDS = [
    'social_core.backends.google.GoogleOAuth2',
    'social_core.backends.facebook.FacebookOAuth2',
    'django.contrib.auth.backends.ModelBackend',
]
# Até aqui
```

Perceba que também adicionamos um novo atributo chamado `AUTHENTICATION_BACKENDS`. Vamos sobrescrever o `AUTHENTICATION_BACKENDS` original do Django injetando nele o `FacebookOAuth2` e `GoogleOAuth2`, assim teremos esses dois mais o `ModelBackend` como modelo de autenticação da aplicação. O `ModelBackend` é o modelo padrão da nossa aplicação que necessita de cadastro de usuário e senha.

Quando realizamos login com alguma rede social, geralmente alguns dados são transitados entre a API da rede social e nossa aplicação. Esses dados podem ser `username`, `email`, `photo` e muitos outros que geralmente configuramos quando habilitamos a API no site da rede social. O `social_django` cria algumas tabelas em nossa aplicação para poder gerenciar esses dados sem interferir na tabela usuário padrão da aplicação.

Por isso, precisamos rodar o comando de migrate na aplicação para criar essas novas tabelas. Rode o comando a seguir em seu terminal, com o `virtualenv` ativado.

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
```

Você terá um resultado parecido com este aqui:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, medicSearch, sessions,
social_django
Running migrations:
  Applying social_django.0001_initial... OK
  Applying social_django.0002_add_related_name... OK
  Applying social_django.0003_alter_email_max_length... OK
  Applying social_django.0004_auto_20160423_0400... OK
  Applying social_django.0005_auto_20160727_2333... OK
  Applying social_django.0006_partial... OK
  Applying social_django.0007_code_timestamp... OK
  Applying social_django.0008_partial_timestamp... OK

(venv) tiago_luiz@ubuntu:~/livro_django$
```

Essas tabelas vão auxiliar o módulo com o controle de cadastro e login por rede social. Com tudo instalado e migrado, já podemos começar a configurar as redes socias que vamos utilizar: Facebook e Google.

12.2 Configurando a url

Vamos começar a configurar nossa url que será usada para realizarmos o login da rede social. Abra o arquivo `AuthUrls.py` dentro da pasta `medicSearch/urls` e altere-o como no exemplo a seguir:

medicSearch/urls/AuthUrls.py

```
# Altere a linha a seguir adicionando o `include` ao import
from django.urls import path, include
from medicSearch.views.AuthView import login_view, register_view,
logout_view

urlpatterns = [
```

```
    path("login", login_view, name='login'),
    path("register", register_view, name='register'),
    path("logout", logout_view, name='logout'),
    # Adicione a linha a seguir
    path('social-auth/', include('social_django.urls',
namespace="social")),
]
```

Botões no formulário

Vamos adicionar todos os botões de login com rede social no sistema e, a cada configuração de rede social, vamos alterar a url que inicialmente ficará vazia.

Abra o arquivo `auth.html` que fica na pasta `medicSearch/templates/auth` e vamos alterá-lo:

medicSearch/templates/auth/auth.html

```
...código oculto
    <a href="{{link_href}}" class="text-center mt-2 mb-2" style="display:
block;">{{link_text}}</a>
    <button type="submit" class="btn btn-primary">{{button_text}}</button>
    <!-- Adicione as linhas a seguir -->
    <p class="text-center mt-2 mb-2" style="display: block;">Ou</p>
    <div class="social-container">
        <a class="btn" id="facebook" href="#"><i class="fa fa-facebook-f">
</i></a>
        <a class="btn" id="google" href="#"><i class="fa fa-google"></i>
</a>
    </div>
    <!-- Até aqui -->
</form>
...código oculto
```

Com esses botões criados nossas telas de login e registro ficarão assim:

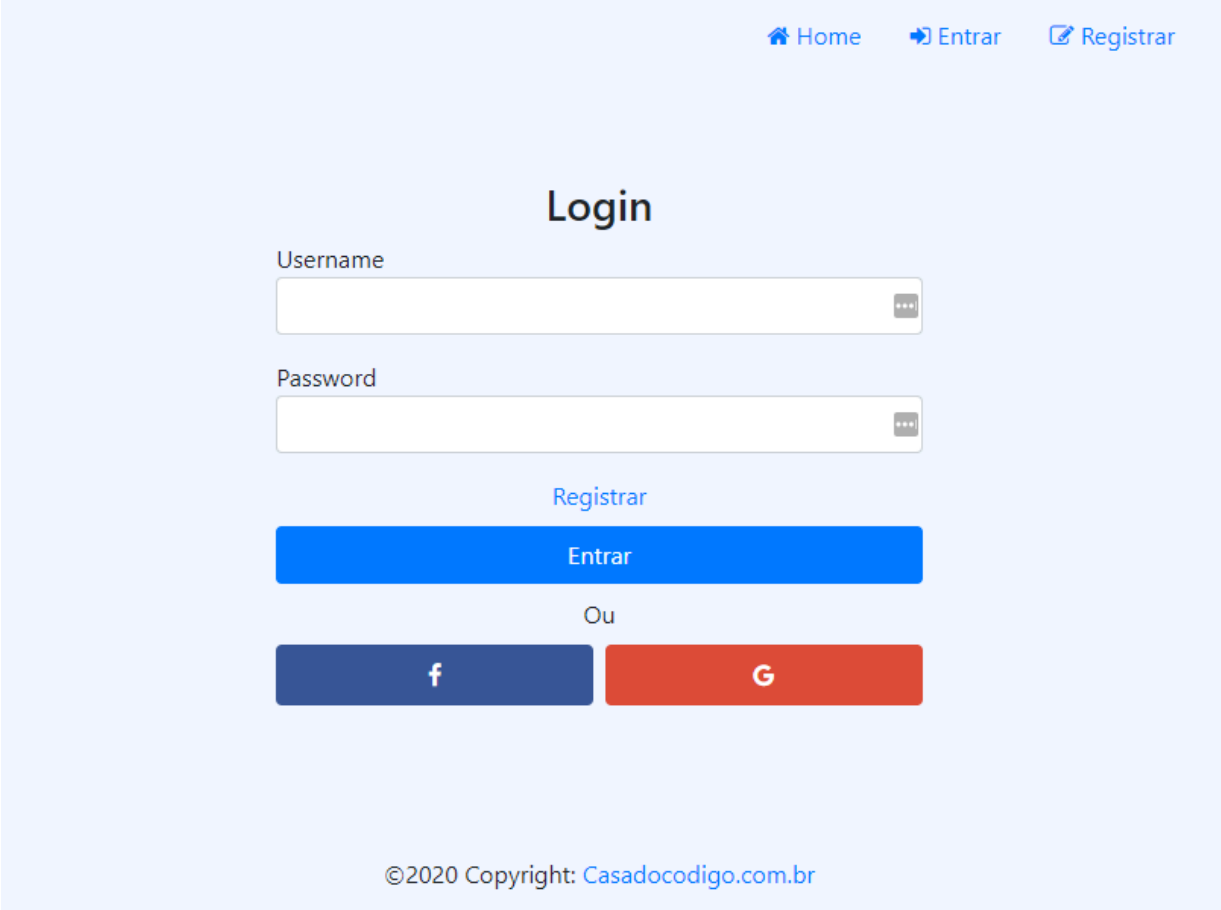


Figura 12.1: Formulário de login - Redes sociais

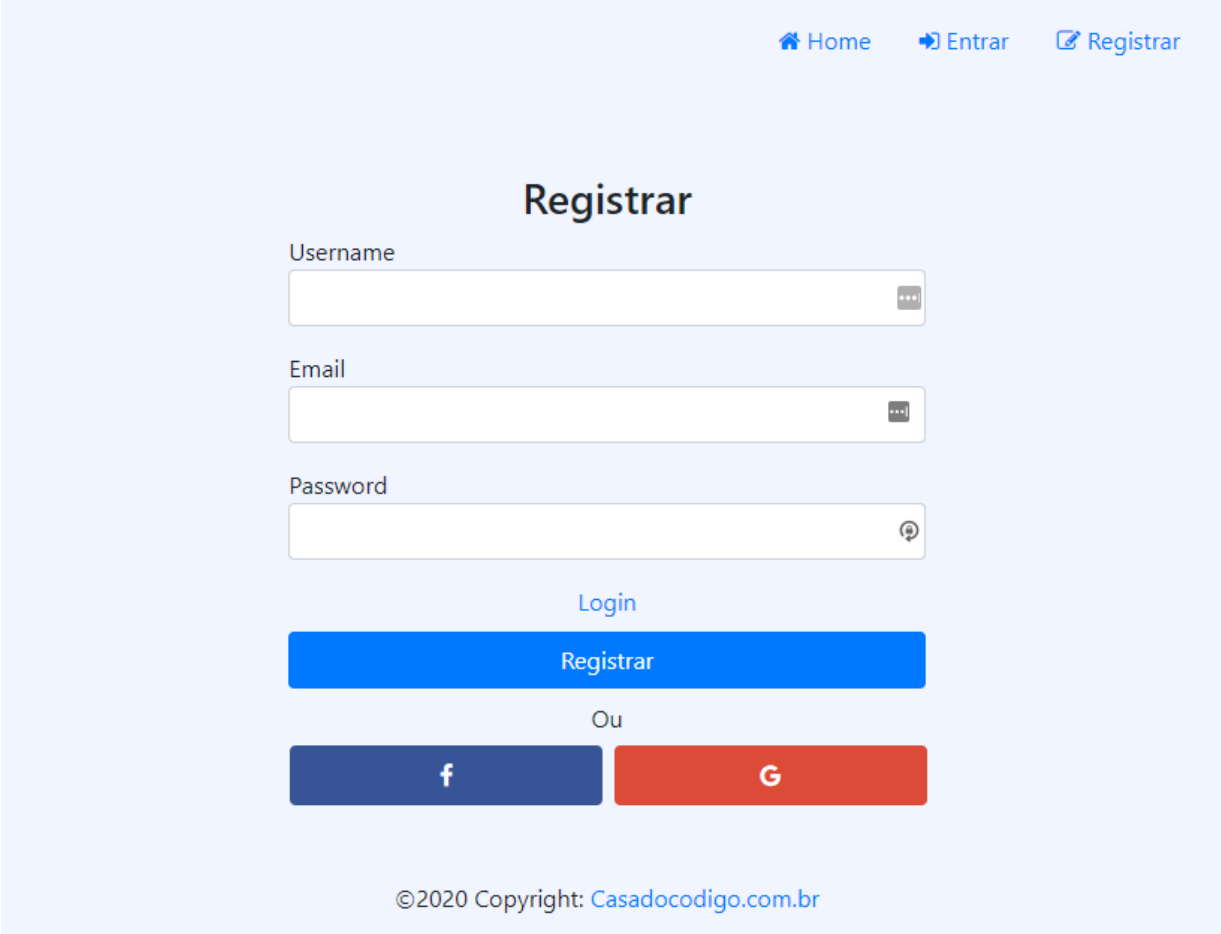


Figura 12.2: Formulário de registro - Redes sociais

Vamos começar a configurar nossas redes sociais!

12.3 Configurando as redes sociais

Agora, precisaremos configurar nossa aplicação em duas etapas para cada rede social:

- Habilitar na rede social a API de autenticação: isso permitirá que nosso sistema solicite à API da rede social as informações necessárias para realizar o login ou registro em nossa plataforma. É necessário para que só tenham permissão de

solicitar login por rede social as aplicações que fizeram a configuração correta para isso.
- Adicionar ao nosso arquivo de configuração os `tokens` : eles serão usados para que nosso sistema acesse a API de cada rede social toda vez que um login social for solicitado. Se nossa aplicação informar o `token` correto, significa que ela está apta a realizar o login.

Facebook

Vamos começar configurando o aplicativo do Facebook. Abra a url https://developers.facebook.com/apps/ e, se não estiver logado, efetue o login. Ao entrar, você verá uma tela com seus aplicativos criados, caso já tenha algum; caso não tenha, verá uma tela vazia com uma opção para criar um app.

Clique na opção adicionar um novo aplicativo:

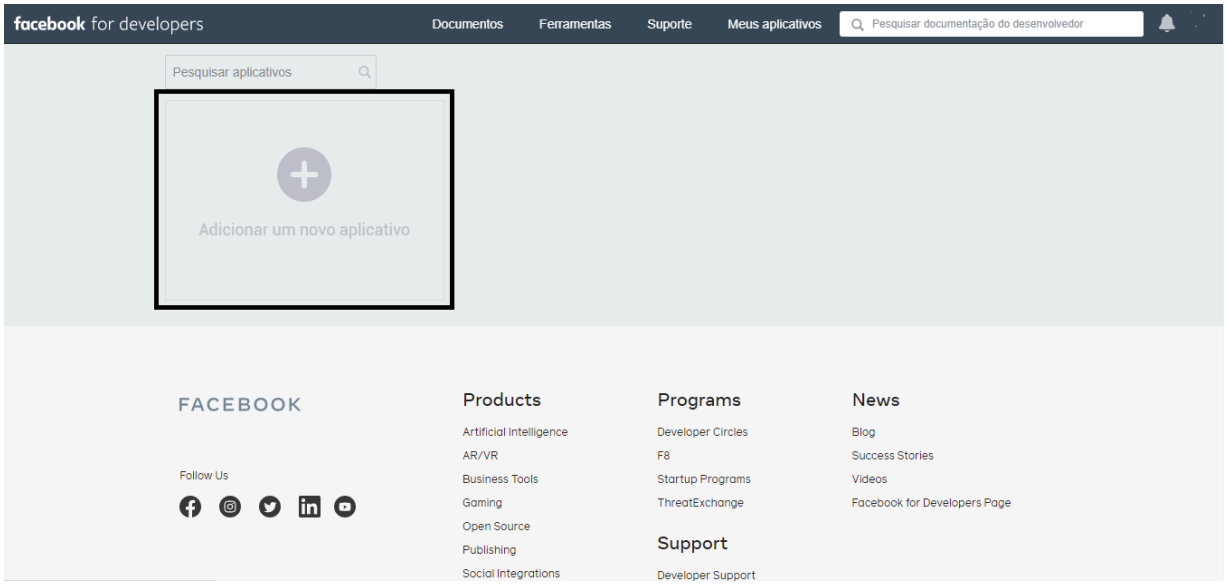


Figura 12.3: Login Facebook - Novo app

Selecione esta opção para mais opções:

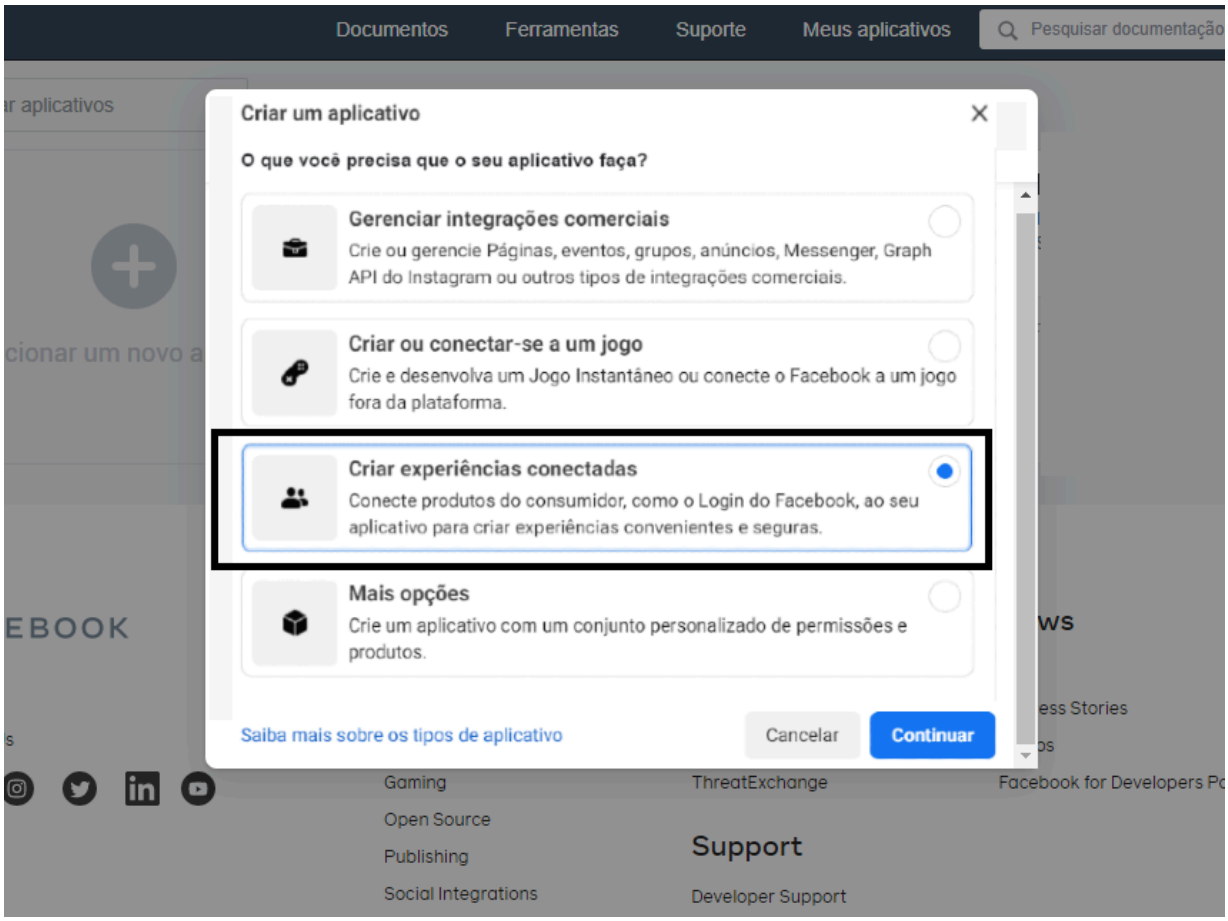


Figura 12.4: Login Facebook - Tipo de aplicativo

Insira um nome no aplicativo, confira se o campo e-mail de contato do aplicativo está preenchido e clique em `Criar ID do aplicativo` :

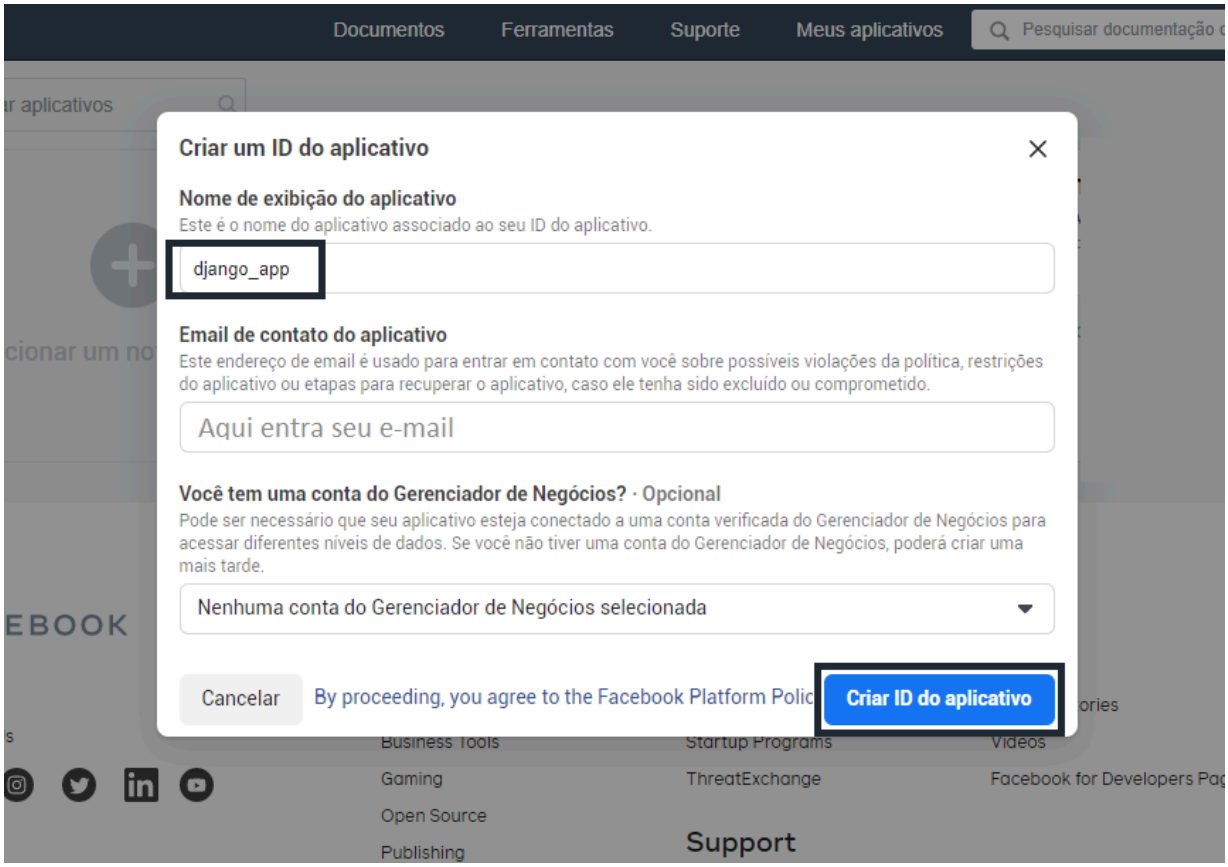


Figura 12.5: Login Facebook - Nome do aplicativo

Às vezes aparece um Recaptcha, clique nele e espere seu app ser criado:

Na lateral direita tem um menu suspenso chamado `Configurações`. Abra-o e clique na opção `Básico`:

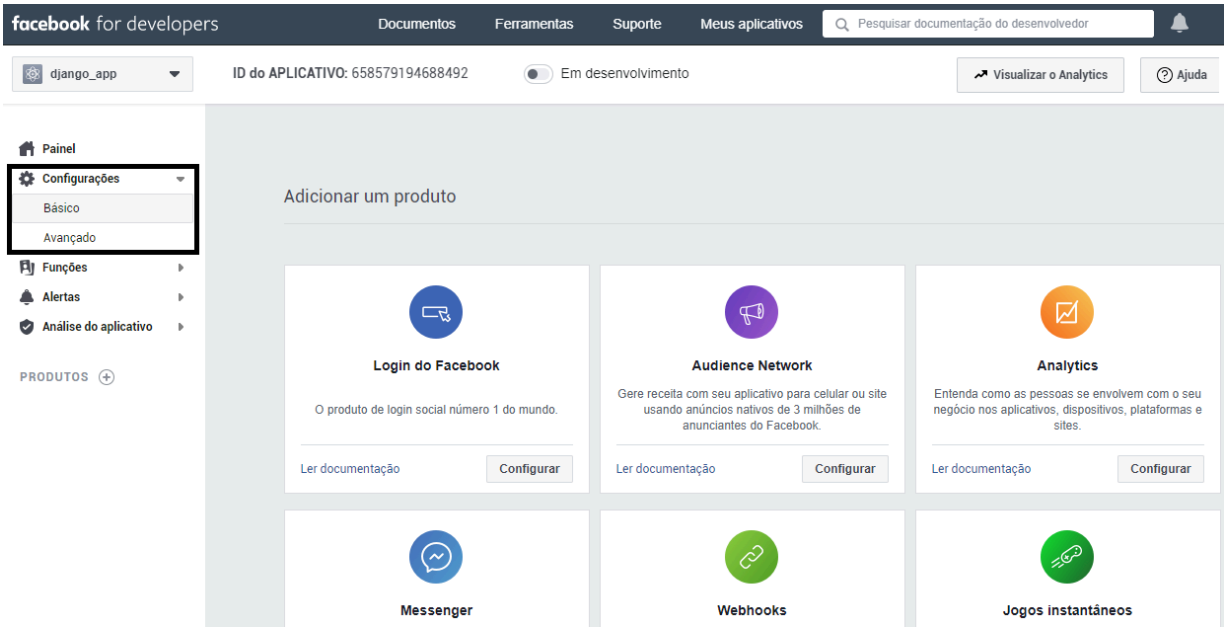


Figura 12.6: Login Facebook - Configurações básicas

Preencha o campo `Domínios do aplicativo` com o valor `localhost`. Não se esqueça de que isso deve ser atualizado quando a aplicação estiver no endereço de domínio da nuvem:

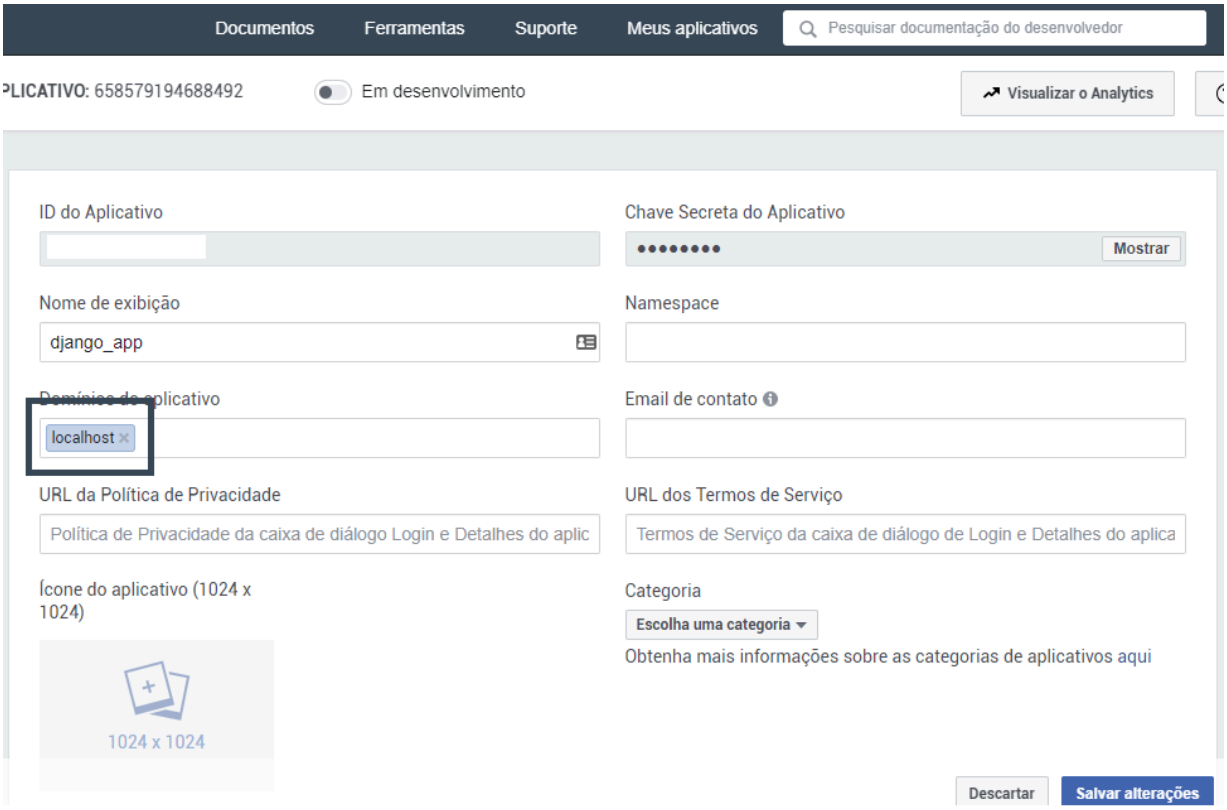


Figura 12.7: Login Facebook - Domínios do aplicativo

Role a página até o final e clique no botão `Adicionar plataforma` :

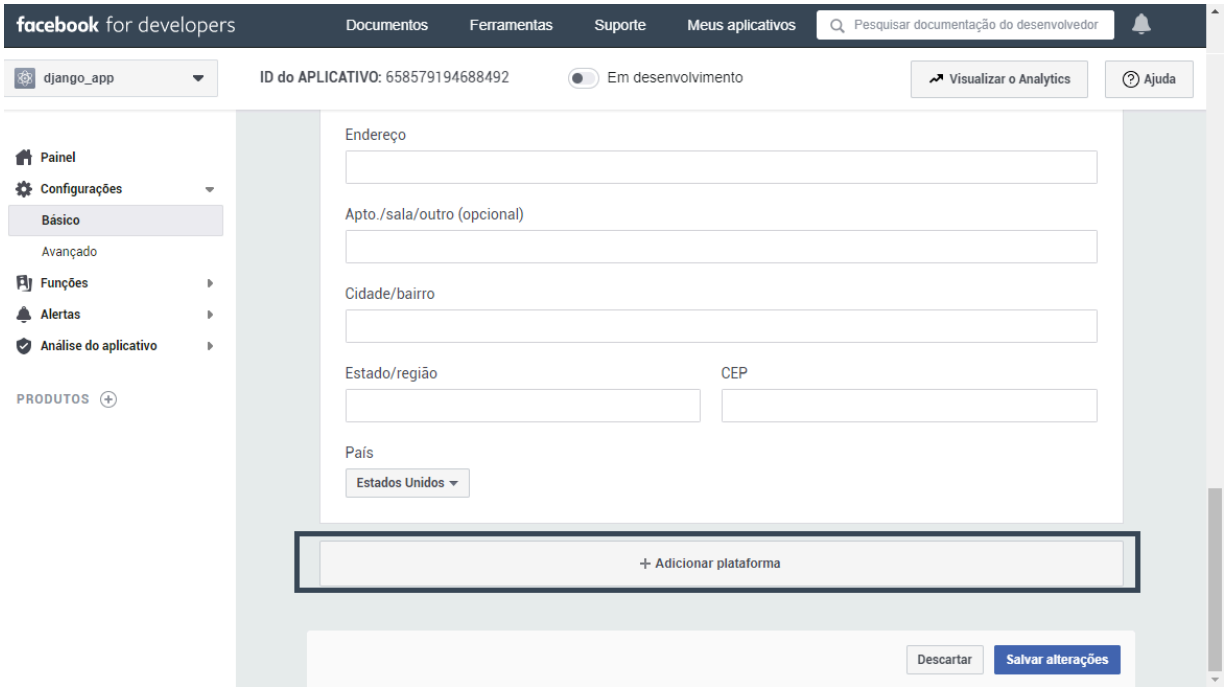


Figura 12.8: Login Facebook - Adicionar plataforma

No modal que abrirá selecione a opção `[Site]` :

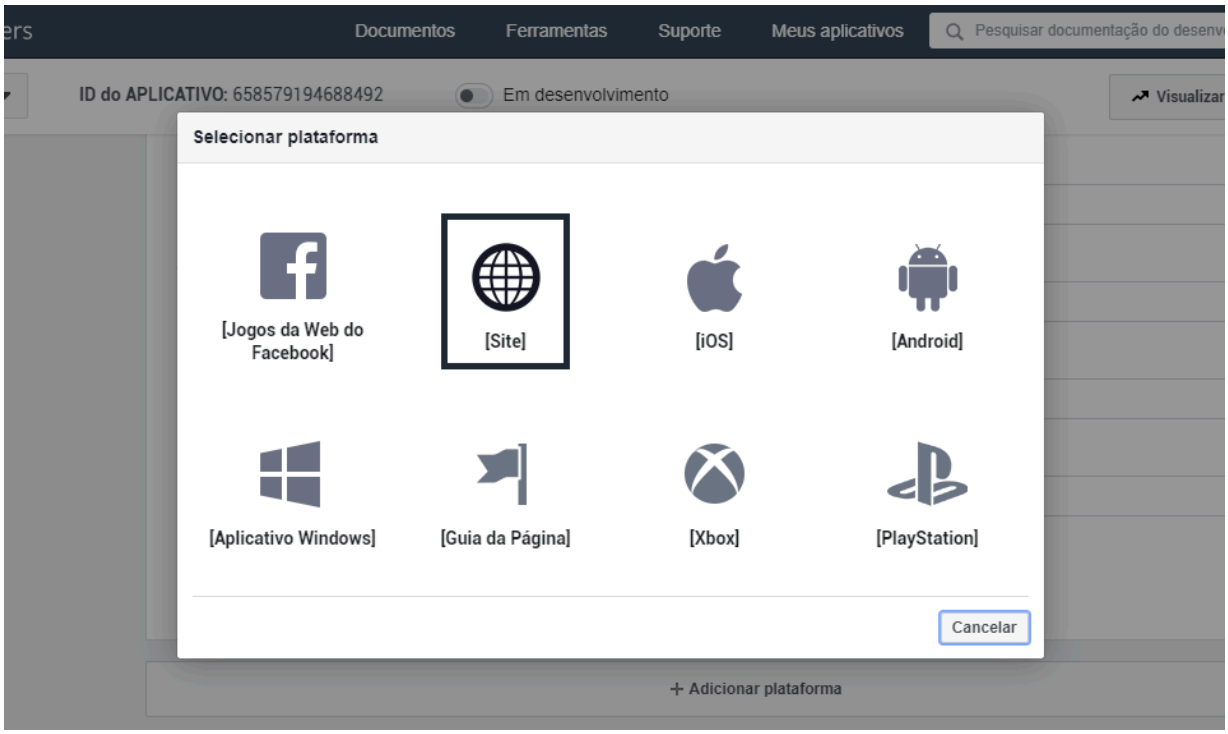


Figura 12.9: Login Facebook - Adicionar plataforma

Preencha o campo `URL do site` com o valor `http://localhost:8000/` (caso seu endereço seja outro, adicione-o no lugar deste) e clique em `Salvar alterações`:

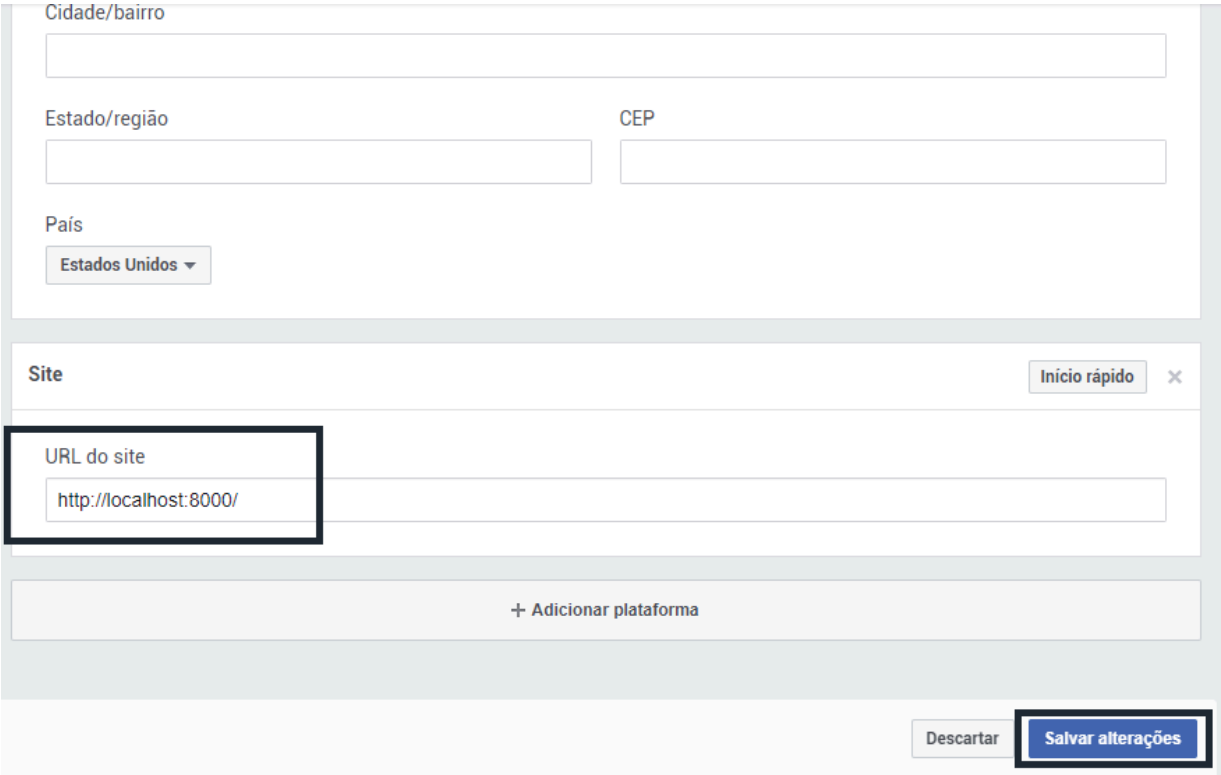


Figura 12.10: Login Facebook - URL do site

Suba para o topo da página novamente e clique em `mostrar` no campo `Chave Secreta do Aplicativo`:

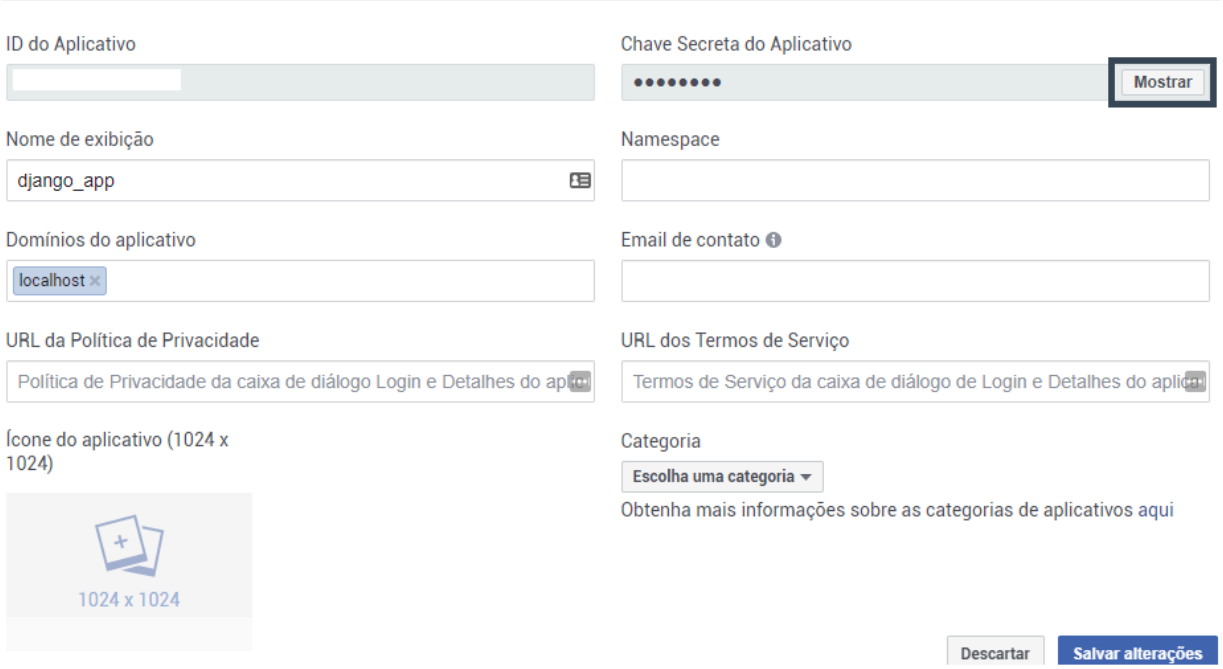


Figura 12.11: Login Facebook - Chave secreta

Ele pedirá sua senha do `Facebook`, preencha para ver a `Chave Secreta do Aplicativo`:

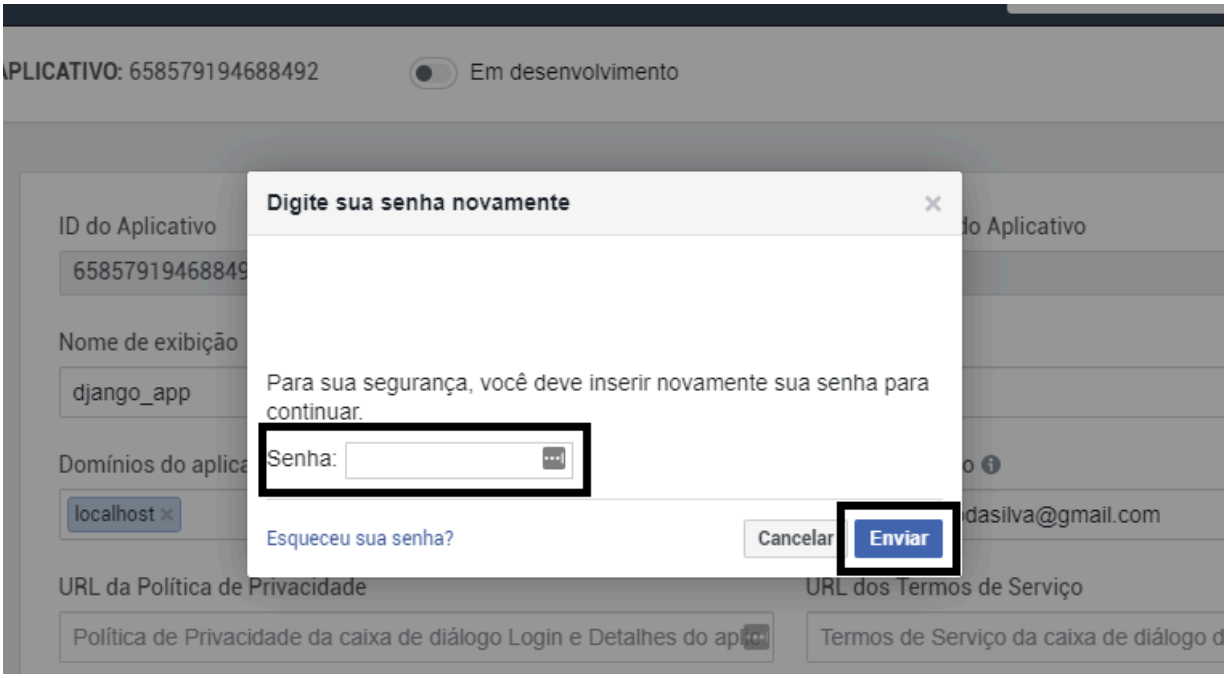


Figura 12.12: Login Facebook - Senha do Facebook

Agora o valor da chave secreta está visível. Anote os valores `ID do Aplicativo` e `Chave Secreta do Aplicativo`, pois precisaremos usá-los em nosso arquivo `settings.py`:

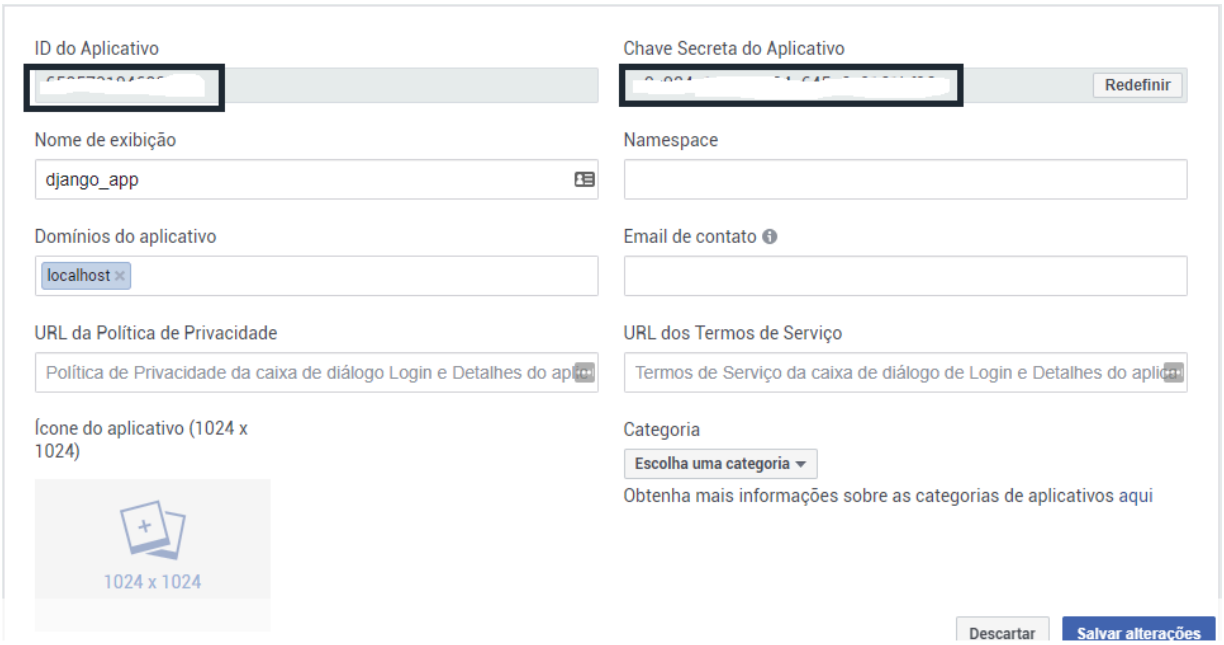


Figura 12.13: Login Facebook - Chaves da API

Abra o arquivo `settings.py` que fica na pasta `medicSearchAdmin/settings` e vamos adicionar a ele o `ID do Aplicativo` e `Chave Secreta do Aplicativo` como no código a seguir:

medicSearchAdmin/settings/settings.py

```
AUTHENTICATION_BACKENDS = [
    'social_core.backends.google.GoogleOAuth2',
    'social_core.backends.facebook.FacebookOAuth2',
    'django.contrib.auth.backends.ModelBackend',
]

# Adicione as linhas a seguir no final do arquivo
SOCIAL_AUTH_FACEBOOK_KEY = "CÓDIGO ID do Aplicativo" # App ID
SOCIAL_AUTH_FACEBOOK_SECRET = "Chave Secreta do Aplicativo" # App Secret
```

Agora vá ao arquivo `auth.html` que fica na pasta `medicSearch/templates/auth` e altere o botão do Facebook para ficar

igual ao do código a seguir:

medicSearch/templates/auth/auth.html

```
<!-- Altere o link para ficar igual ao da linha a seguir -->
<a class="btn" id="facebook" href="{% url 'social:begin' 'facebook' %}"><i
class="fa fa-facebook-f"></i></a>
```

Se você tentar realizar login agora com o Facebook já vai conseguir. Perceba que um usuário foi criado no painel de administração com o primeiro e o último nome, ainda sem e-mail, mas já resolveremos isso.

Se acessarmos a url do admin (estando logado como admin), veremos um painel novo lá embaixo, que é para o módulo `social-auth-app-django` gerenciar o vínculo entre os usuários que foram cadastrados na plataforma pela rede social:

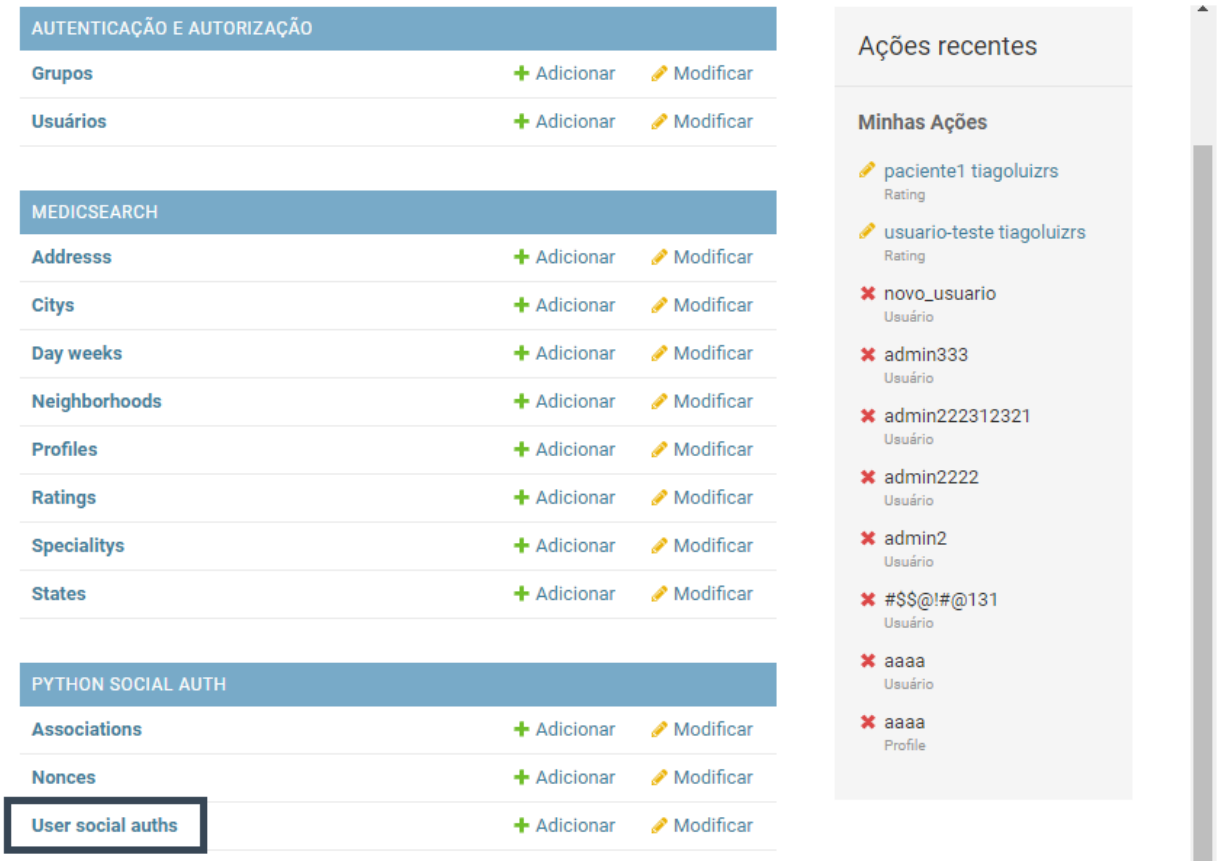


Figura 12.14: Admin Social

Note que, se clicarmos na opção `User social auths`, veremos os usuários que foram cadastrados com rede social e o `uid` dele é o id da rede social. Essa tabela tem relacionamento com a tabela `user` do Django:

Figura 12.15: Admin Social - Usuário UID

Agora vamos ver como solicitar alguns dados a mais para o Facebook como a imagem do perfil, por exemplo.

Dados adicionais Facebook

Abra o arquivo `settings.py` que fica na pasta `medicSearchAdmin/settings` e vamos fazer algumas modificações. Vou mostrar somente as áreas que devem ser alteradas, porque o código é extenso e a alteração ocorre em áreas distintas do código:

medicSearchAdmin/settings/settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
```

```
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                # Adicione as linhas a seguir
                'social_django.context_processors.backends', # Adicione
isso
                'social_django.context_processors.login_redirect', #
Adicione isso
            ],
        },
    },
]

... código oculto

# Desça até o final do arquivo para ver o código a seguir
SOCIAL_AUTH_FACEBOOK_KEY = "CÓDIGO ID do Aplicativo" # App ID
SOCIAL_AUTH_FACEBOOK_SECRET = "Chave Secreta do Aplicativo" # App Secret
# Adicione as linhas a seguir
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email', 'user_link'] # add this
SOCIAL_AUTH_FACEBOOK_PROFILE_EXTRA_PARAMS = {       # add this
  'fields': 'id, name, email, picture.type(large), link'
}
SOCIAL_AUTH_FACEBOOK_EXTRA_DATA = [                 # add this
    ('name', 'name'),
    ('email', 'email'),
    ('picture', 'picture'),
    ('link', 'profile_url'),
]
```

Vamos entender o que cada um faz:

- **SOCIAL_AUTH_FACEBOOK_SCOPE**: contém uma lista de permissões para acessar as propriedades de dados que nosso aplicativo requer.
- **SOCIAL_AUTH_FACEBOOK_PROFILE_EXTRA_PARAMS**: possui uma chave `fields` em que o valor é uma lista de atributos que devem ser retornados pelo Facebook quando o usuário tiver efetuado login com êxito. Os valores dependem de `SOCIAL_AUTH_FACEBOOK_SCOPE`.

- **SOCIAL_AUTH_FACEBOOK_EXTRA_DATA**: precisamos especificá-lo para armazenar os dados adicionais que solicitamos ao banco de dados.

Se tentarmos fazer login novamente na plataforma veremos uma tela assim:

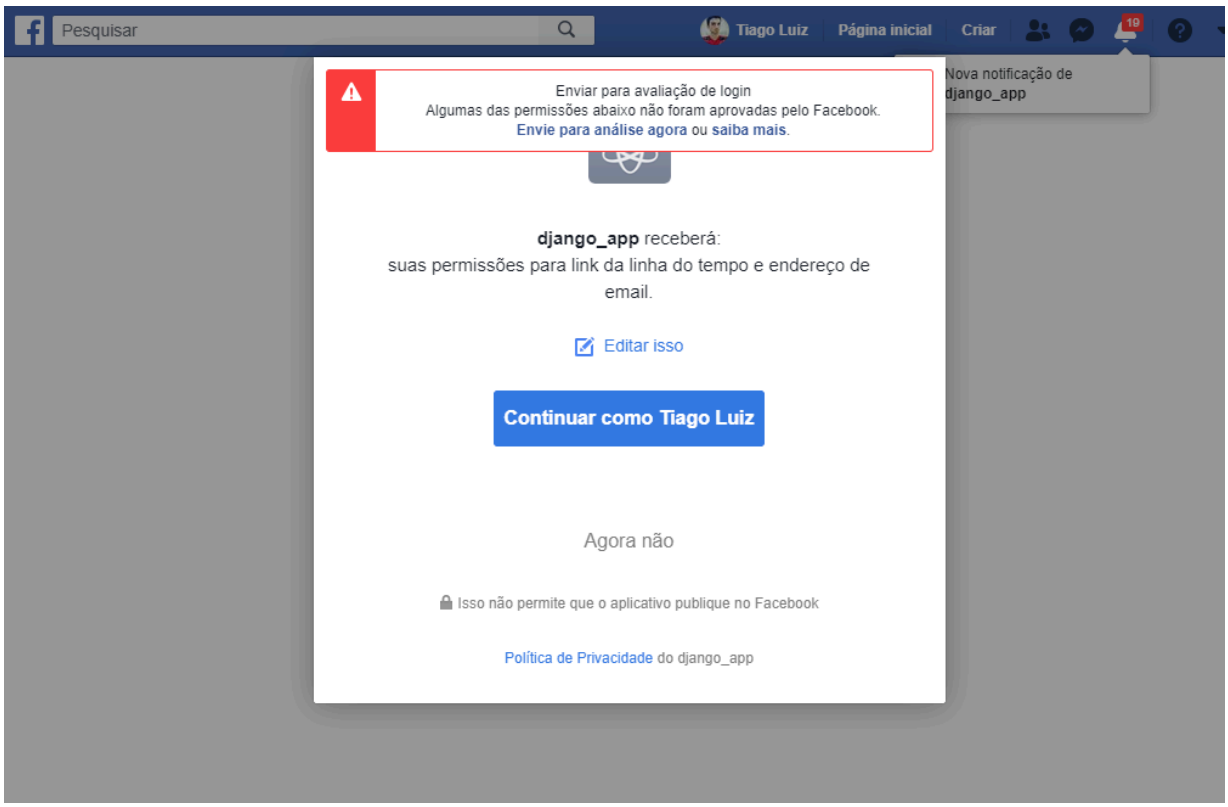


Figura 12.16: Login Facebook - Extra fields

Ao fazer login, se olharmos o campo `extra data` do painel `user social auths` veremos que já trazemos os dados que foram solicitados (não se esqueça de fazer logout no perfil do Facebook e entrar como admin):

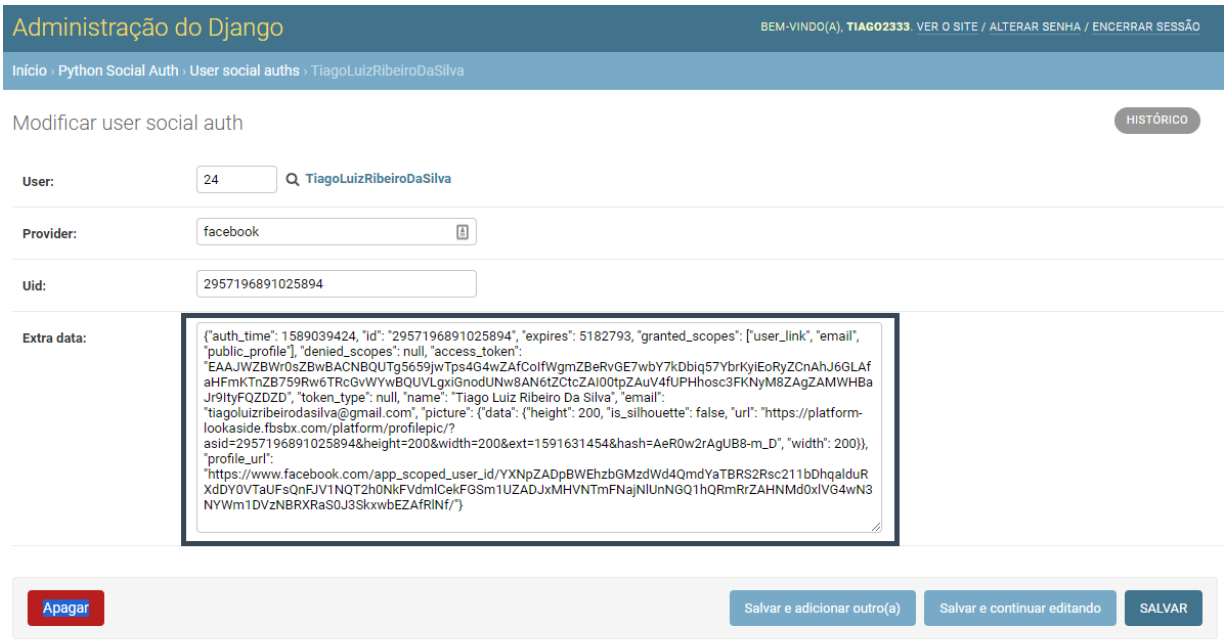


Figura 12.17: Admin - Extra fields

Podemos utilizar esses dados caso precisemos. Se quisermos, por exemplo, acessar a foto do Facebook deste usuário, é só acessarmos o campo `extra fields`. Se quisermos usar qualquer atributo que está no campo `extra fields` em nosso template `html` podemos fazê-lo chamando `{{ass.extra_data.nome_do_campo}}`. E se quisermos chamar a imagem, fazemos `{{ass.extra_data.picture.data.url}}`. Simples, não é mesmo?

Agora veremos como realizar a configuração do login usando o Google.

Google

Para configurar o aplicativo do Google, abra a url `https://console.developers.google.com/projectcreate`. Uma tela para a criação de um app será exibida.

Adicione um nome para seu app e clique em `Salvar`:

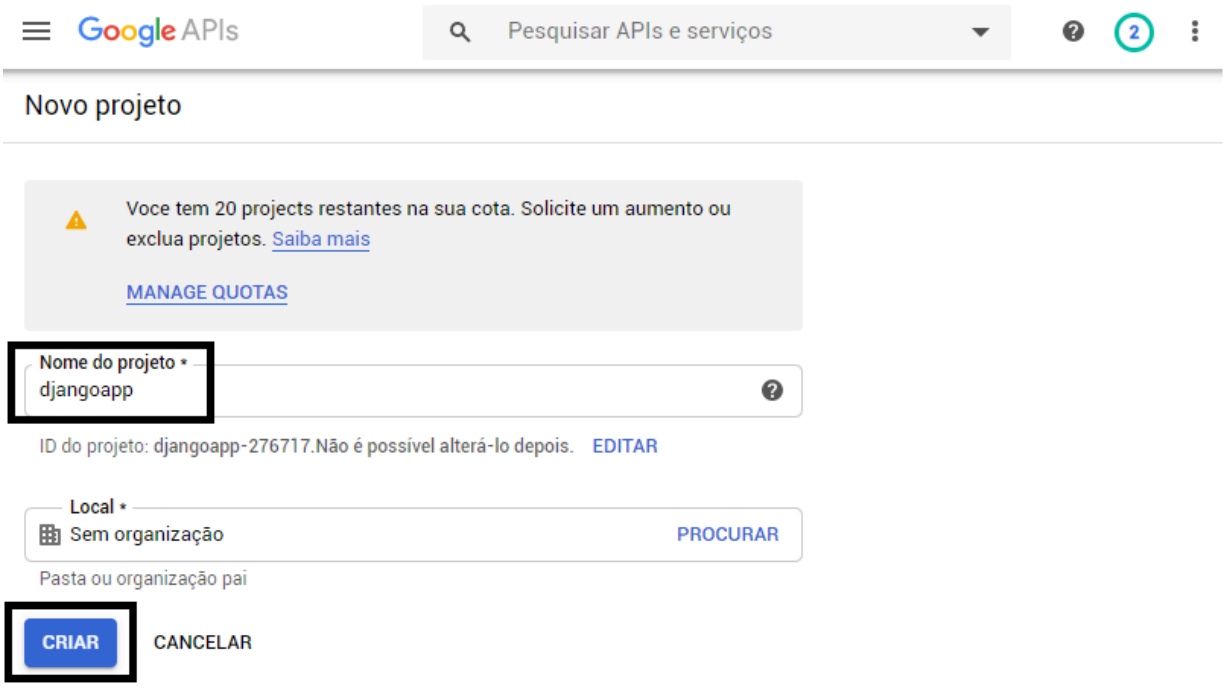


Figura 12.18: Login Google - Criar novo app

Após criar o projeto, não esqueça de selecioná-lo, pois se existir algum outro projeto em seu Google console, ele poderá estar selecionado por padrão. Selecione o novo projeto antes de continuar.

Adicione um nome para seu app e clique em `Salvar`:

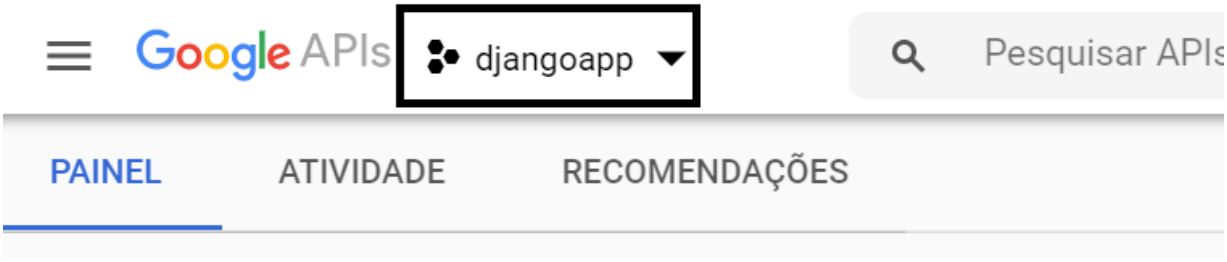


Figura 12.19: Login Google - Criar novo app

Acesse https://console.developers.google.com/apis/credentials e clique no botão `CONFIGURAR TELA DE CONSENTIMENTO`:

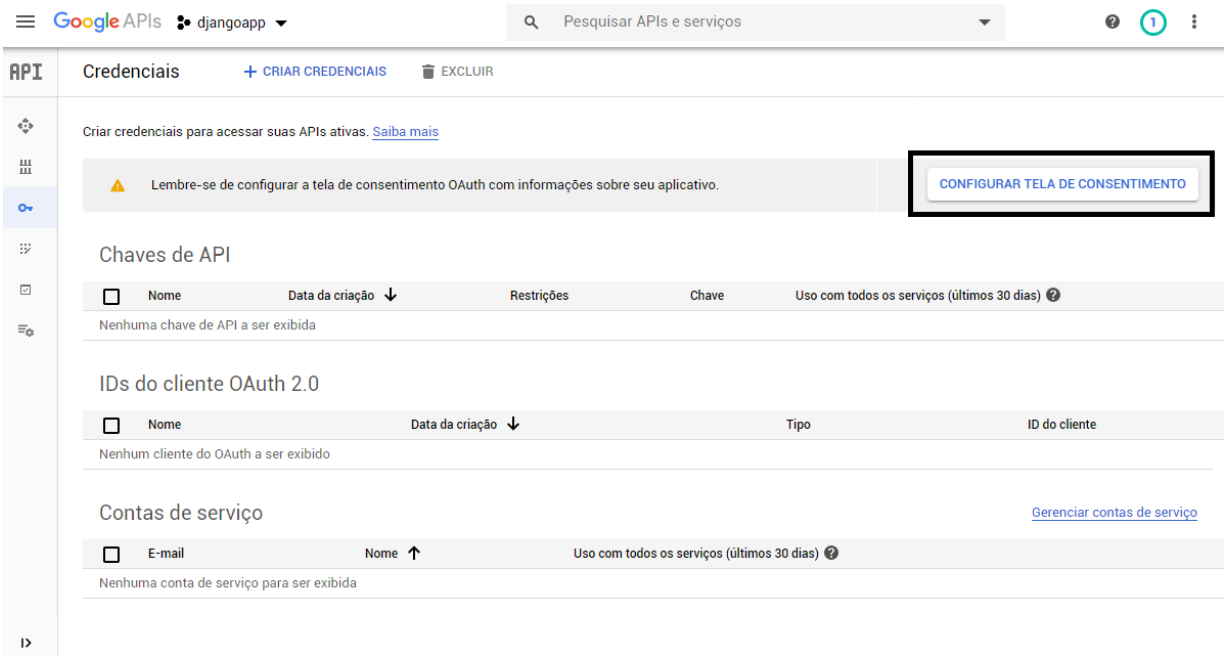


Figura 12.20: Login Google - Tela de consentimento 1

Selecione a opção `Externo` e clique em `Criar`:

Figura 12.21: Login Google - Tela de consentimento 2

Insira um nome e um e-mail para o app, role a tela para baixo, onde também terá um campo de e-mail para preencher, que será o último

campo do formulário. Após isso clique em salvar:

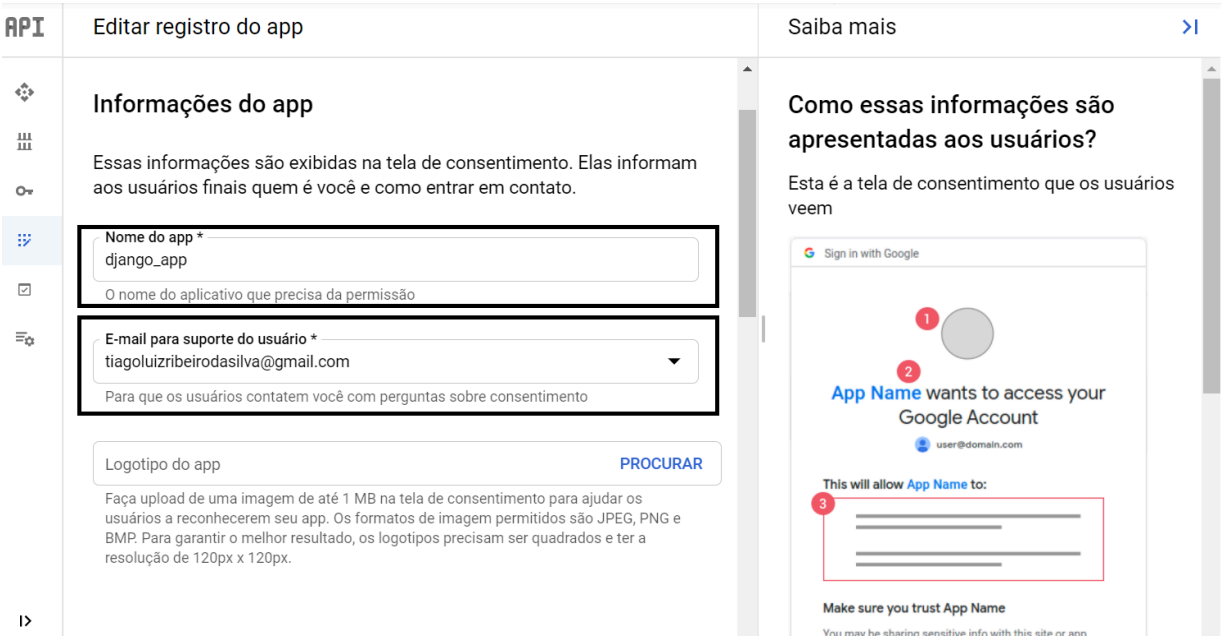


Figura 12.22: Login Google - Tela de consentimento 3

Acesse mais uma vez a url https://console.developers.google.com/apis/credentials e clique no botão `CRIAR CREDENCIAIS`. Uma janela se abrirá, clique na opção `ID do client OAuth`:

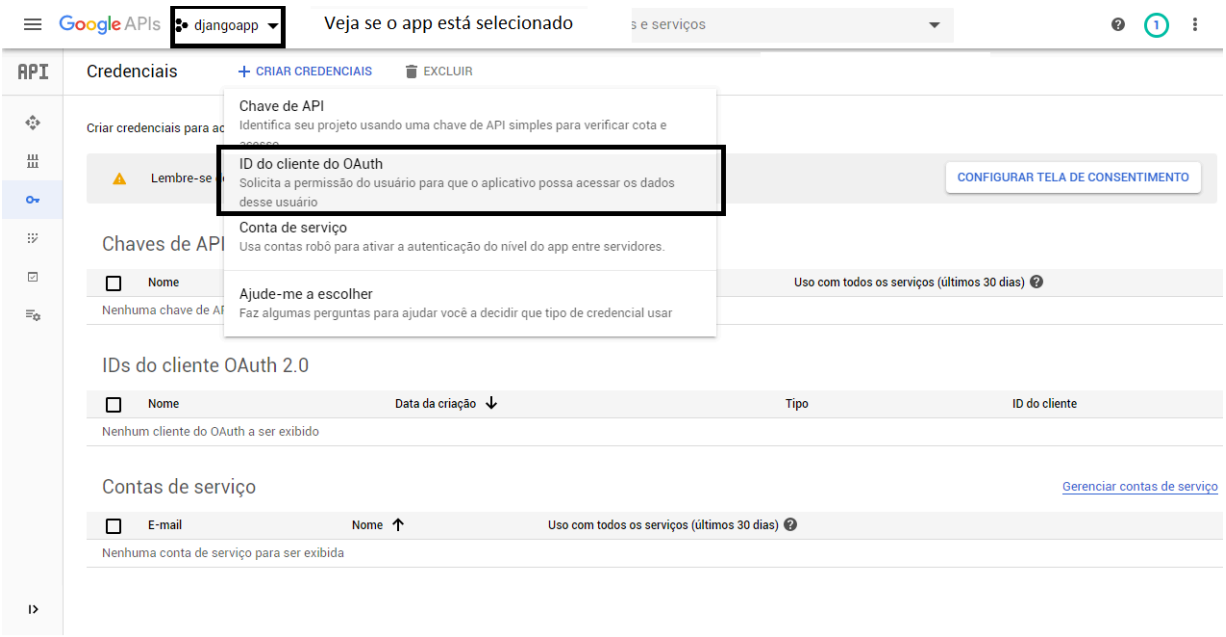


Figura 12.23: Login Google - Criar credencial

Nessa tela, siga os seguintes passos:

- Marque a opção `Aplicativo web`;
- Adicione um nome para o aplicativo;
- Adicione no campo `URLs de redirecionamento autorizados` o valor `http://localhost:8000/social-auth/complete/google-oauth2/`. Fique atento, quando a aplicação estiver na url original, você vai precisar mudar esse valor para o correspondente ao domínio. Coloque exatamente como está aqui, não remova nem a barra final, pois pode dar problema na configuração;
- Clique em `criar`.

Figura 12.24: Login Google - Configurar credencial

- Uma janela se abrirá, anote as credenciais que serão geradas, vamos usá-las no arquivo `settings.py` :

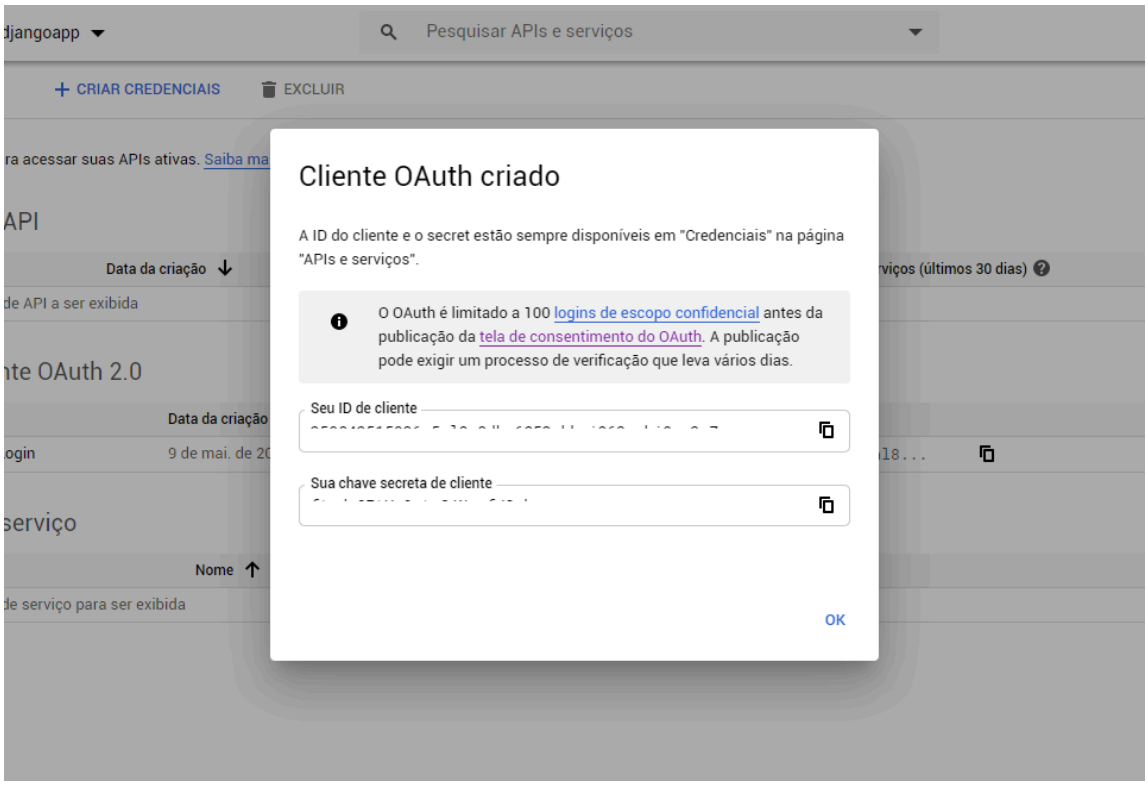


Figura 12.25: Login Google - Configurar credencial

- Anote as chaves que você recebeu, porque precisaremos habilitar mais um recurso no painel do Google antes de ir para o arquivo `settings.py`.
- Abra https://console.developers.google.com/apis/library, pesquise pela biblioteca chamada `Google+ API` e clique nela:

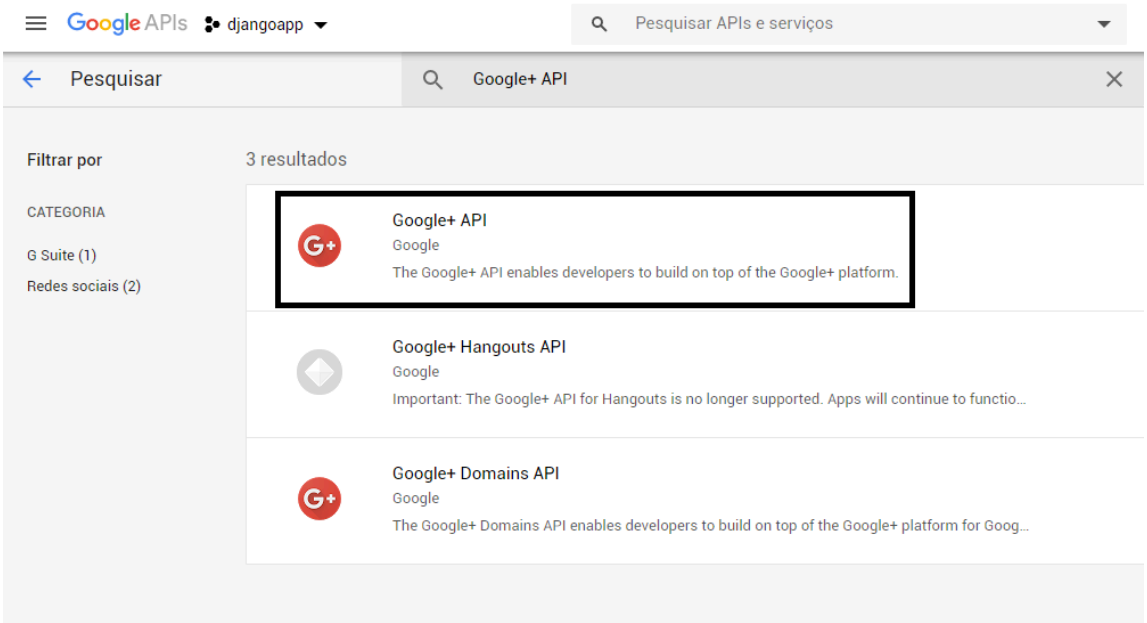


Figura 12.26: Login Google - Google+ API

- Ao clicar nessa biblioteca, você verá um botão `ATIVAR`. Clique nele. Essa é a API do Google que permitirá que façamos o login na aplicação usando a API OAuth que criamos nos passos anteriores:

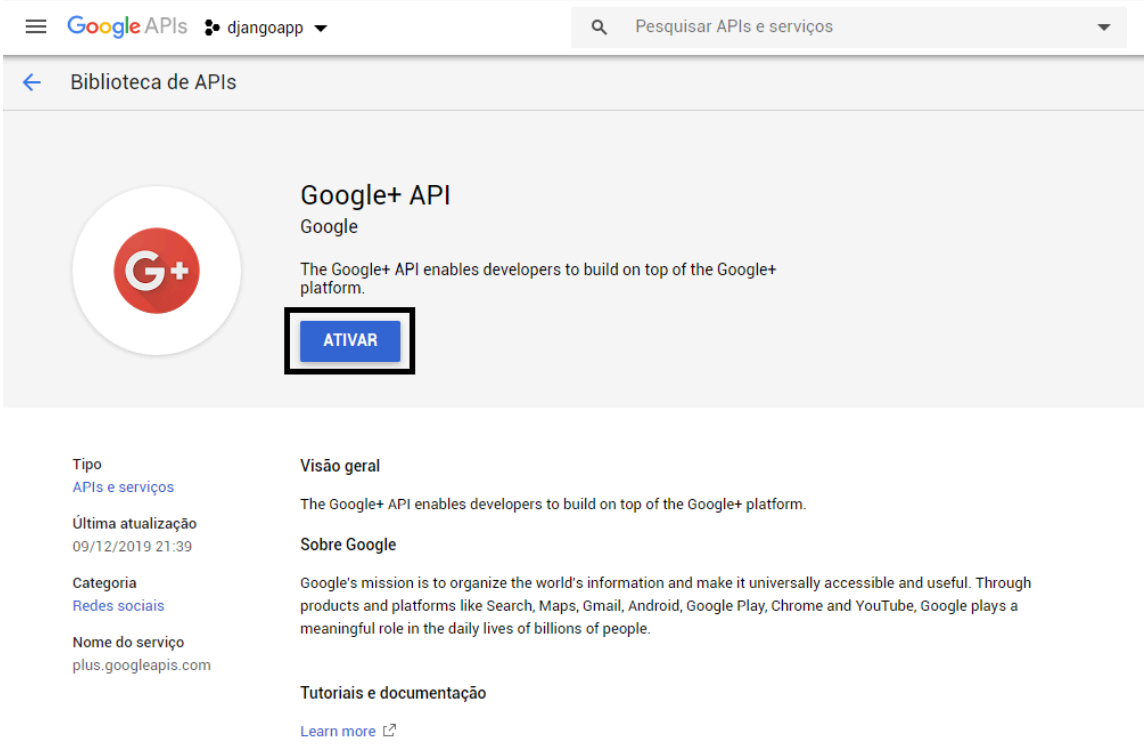


Figura 12.27: Login Google - Google+ API Ativar

Com as credenciais criadas, vamos abrir o arquivo `settings.py` que fica na pasta `medicSearchAdmin/settings` e vamos adicionar as credenciais do Google:

medicSearchAdmin/settings/settings.py

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'Seu ID de cliente'
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'Sua chave secreta de cliente'
```

Agora, abra o arquivo `auth.html`, que fica na pasta `medicSearch/templates/auth`, e altere o botão de login do Google:

medicSearch/templates/auth/auth.html

```
<!-- Altere o link para ficar igual ao da linha a seguir -->
<a class="btn" id="google" href="{% url 'social:begin' 'google-oauth2'
%}"><i class="fa fa-google"></i></a>
```

Com isso, se você tentar realizar o login pelo Google ele já vai funcionar:

Figura 12.28: Login Google - Teste

Ao realizar o login vá até o painel de edição de perfil e veja que o login ocorreu corretamente:

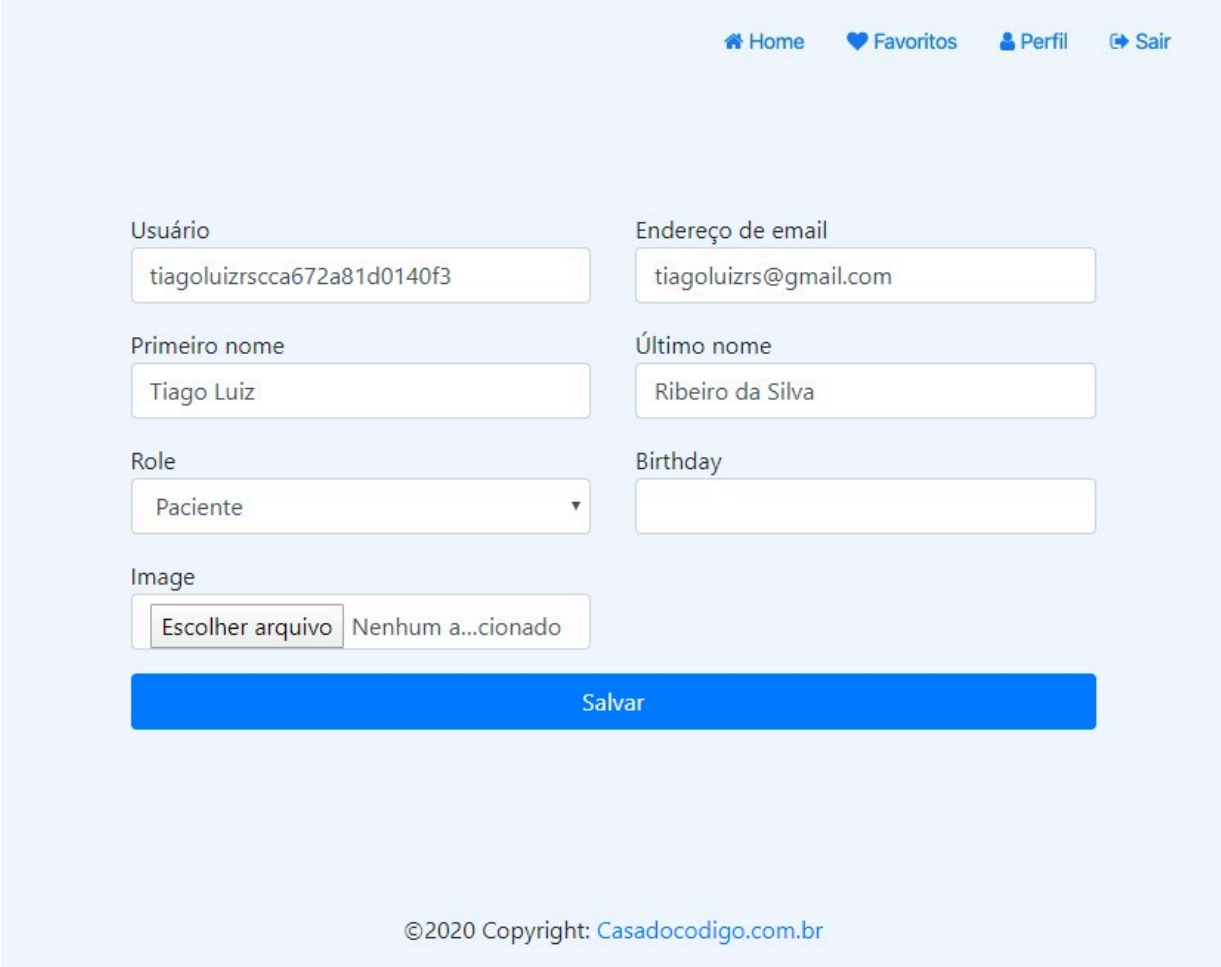


Figura 12.29: Login Google - Perfil

> Em alguns casos, leva até uns 10 minutos para que as configurações propaguem.

Pronto, nossa aplicação consegue realizar login utilizando redes sociais. Esse módulo do Django dá suporte para centenas de outras redes sociais que possuem configurações semelhantes às que vimos aqui.

No próximo capítulo, falaremos sobre serviços de e-mail para confirmação de cadastro e recuperação de senha.

Capítulo 13
Serviços de e-mail

Nesta etapa, aprenderemos a trabalhar com serviços de e-mail, desenvolvendo a recuperação de senha com o serviço que o próprio Django possui. Ao solicitar a redefinição de senha, o sistema armazenará uma hash no campo token da tabela Profile e enviará um link por e-mail com esse valor de token. Com isso, será possível tanto atualizar senha, como confirmar e-mail.

13.1 View de recuperação de senha

Vamos começar criando nosso formulário de recuperação de senha. Abra o arquivo `AuthForm.py` que fica na pasta `medicSearch/forms` e adicione o código a seguir ao final dele:

medicSearch/forms/AuthForm.py

```
from django import forms

class LoginForm(forms.Form):
    ...código oculto não apague

class RegisterForm(forms.Form):
    ...código oculto não apague

# Adicione a classe a seguir no final do arquivo
class RecoveryForm(forms.Form):
    email = forms.CharField(required=True, widget=forms.EmailInput(attrs=
{'class': 'form-control'}))
```

Agora que temos nossa classe de formulário criada, precisamos criar nossa `view` que renderizará a recuperação de senha. Abra o arquivo

`AuthView.py` que fica na pasta `medicSearch/views` e vamos adicionar um novo método que será a `view` de recuperação de senha:

medicSearch/views/AuthView.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.models import User
# Import a model Profile
from medicSearch.models.Profile import Profile
# Import a classe RecoveryForm na linha a seguir
from medicSearch.forms.AuthForm import LoginForm, RegisterForm,
RecoveryForm

...código oculto não apague nada

def logout_view(request):
    logout(request)
    return redirect('/login')

# Adicione os métodos a seguir no final do arquivo
def recovery_view(request):
    recoveryForm = RecoveryForm()
    message = None

    if request.method == 'POST':
        recoveryForm = RecoveryForm(request.POST)

        if recoveryForm.is_valid():
            email = request.POST['email']
            profile = Profile.objects.filter(user__email=email).first()
            if profile is not None:
                try:
                    send_email(profile)
                    message = {
                        'type': 'success',
                        'text': 'Um e-mail foi enviado para sua caixa de
entrada.'
                    }
```

```
                except:
                    message = { 'type': 'danger', 'text': 'Erro no envio
do e-mail.' }
            else:
                message = { 'type': 'danger', 'text': 'E-mail
inexistente.' }
        else:
            message = { 'type': 'danger', 'text': 'Formulário inválido' }

    context = {
        'form': recoveryForm,
        'message': message,
        'title': 'Recuperar senha',
        'button_text': 'Recuperar',
        'link_text': 'Login',
        'link_href': '/login'
    }
    return render(request, template_name='auth/auth.html',
context=context, status=200)

def send_email(profile):
    pass
```

Perceba que por enquanto ainda não temos nada de diferente em nossa `view`. Usaremos o arquivo `auth.html` como template e estamos seguindo o mesmo padrão que o `register` e o `login`. Criamos um método `send_email`, o método de envio de e-mail. Mas antes disso, vamos criar nossa `url` e configurar nosso serviço de e-mail para que tudo ocorra corretamente.

Abra o arquivo `AuthUrls.py` que fica na pasta `medicSearch/urls` e adicione a nova url:

medicSearch/urls/AuthUrls.py

```
from django.urls import path, include
# Import o método recovery_view
from medicSearch.views.AuthView import login_view, register_view,
logout_view, recovery_view
```

```
urlpatterns = [
    path("login", login_view, name='login'),
    path("register", register_view, name='register'),
    path("logout", logout_view, name='logout'),
    # Adicione a linha a seguir
    path("recovery", recovery_view, name='recovery'),
    # Até aqui
    path('social-auth/', include('social_django.urls',
namespace="social")),
]
```

Pronto. Agora precisamos fazer uma pequena alteração no template `auth.html`, que fica na pasta `medicSearch/templates/auth`, para que ele tenha a url da página de recuperação caso esteja na página login. Vamos lá:

medicSearch/templates/auth/auth.html

```
... código oculto não apague nada a seguir
    <a href="{{link_href}}" style="
                    display: block;
                    text-align: center;
                    margin-bottom: 20px;
                ">{{link_text}}</a>
    <!-- Adicione essas linhas após o `link_text` -->
    {% if title == 'Login'%}
    <a href={% url 'recovery' %} class="text-center mt-2 mb-2"
style="display: block;">Esqueci minha senha</a>
    {% endif %}
    <!-- Até aqui -->

    <button type="submit" class="btn btn-primary">{{button_text}}</button>

    <!-- Adicione essa linha após o botão submit -->
    {% if title != 'Recuperar senha' and title != 'Alterar senha' %}
    <!-- Até aqui -->
    <p class="text-center mt-2 mb-2" style="display: block;">Ou</p>
    <div class="social-container">
        <a class="btn" id="facebook" href="{% url 'social:begin'
```

```
'facebook' %}"><i class="fa fa-facebook-f"></i></a>
        <a class="btn" id="google" href="{% url 'social:begin' 'google-
oauth2' %}"><i class="fa fa-google"></i></a>
    </div>
    <!-- Adicione essa linha para fechar o IF acima -->
    {% endif %}
    <!-- Até aqui -->
... código oculto não apague nada a seguir
```

O que fizemos foi adicionar o link de recuperar senha à página de `Login` e ocultar os links de login com rede social nas páginas `Recuperar Senha` e `Alterar senha`.

13.2 Configurando o serviço de e-mail

Temos a tela de configuração de senha em perfeito estado e pronta para funcionar. A única coisa que ela ainda não faz é enviar o e-mail com o token, já que ainda não temos as configurações de envio de e-mail completas. Vamos fazer a configuração do serviço de e-mail a seguir.

Habilitando o Gmail

Se você for utilizar seu Gmail como serviço de envio, precisará ativar o recurso `Acesso a app menos seguro` para que o e-mail possa funcionar em nossa aplicação. Isso ocorre porque o Gmail, por padrão, não permite que os e-mails dele façam envios por ambientes externos a ele. Acesse a url https://myaccount.google.com/security e siga os passos a seguir:

Vá até a parte da página `Acesso a app menos seguro` e clique em `Ativar acesso`:

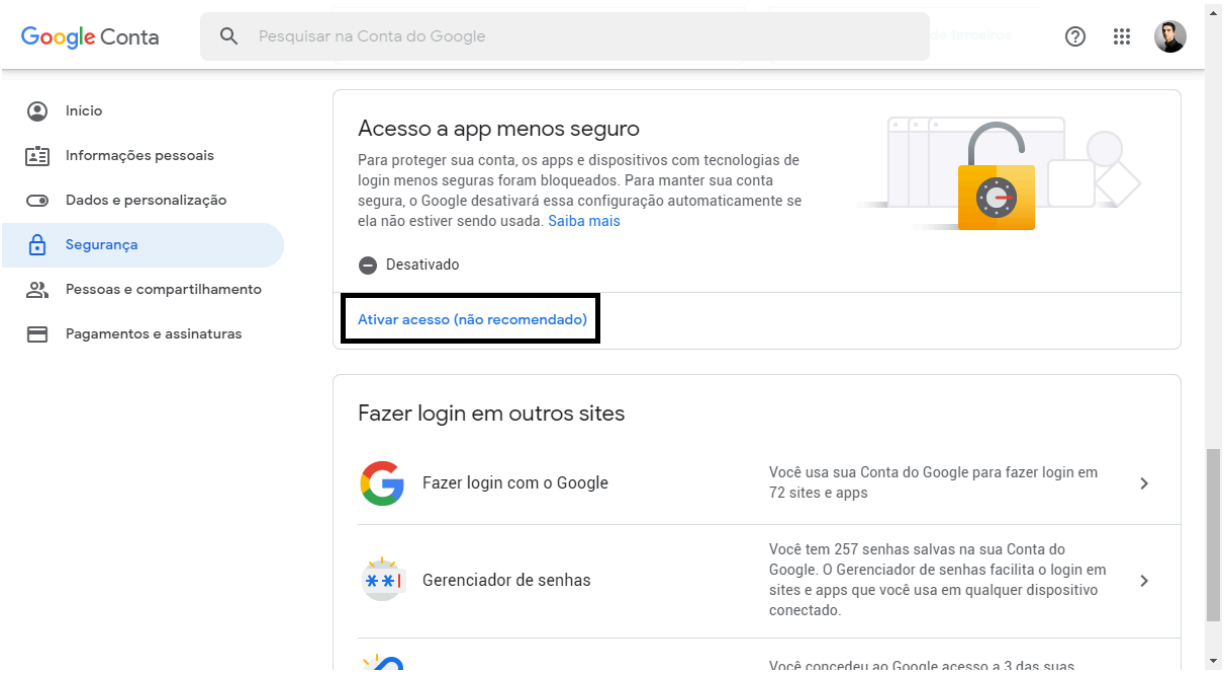


Figura 13.1: Ativar apps menos seguros - Habilitação

Clique na alavanca marcada como na imagem para ativar:

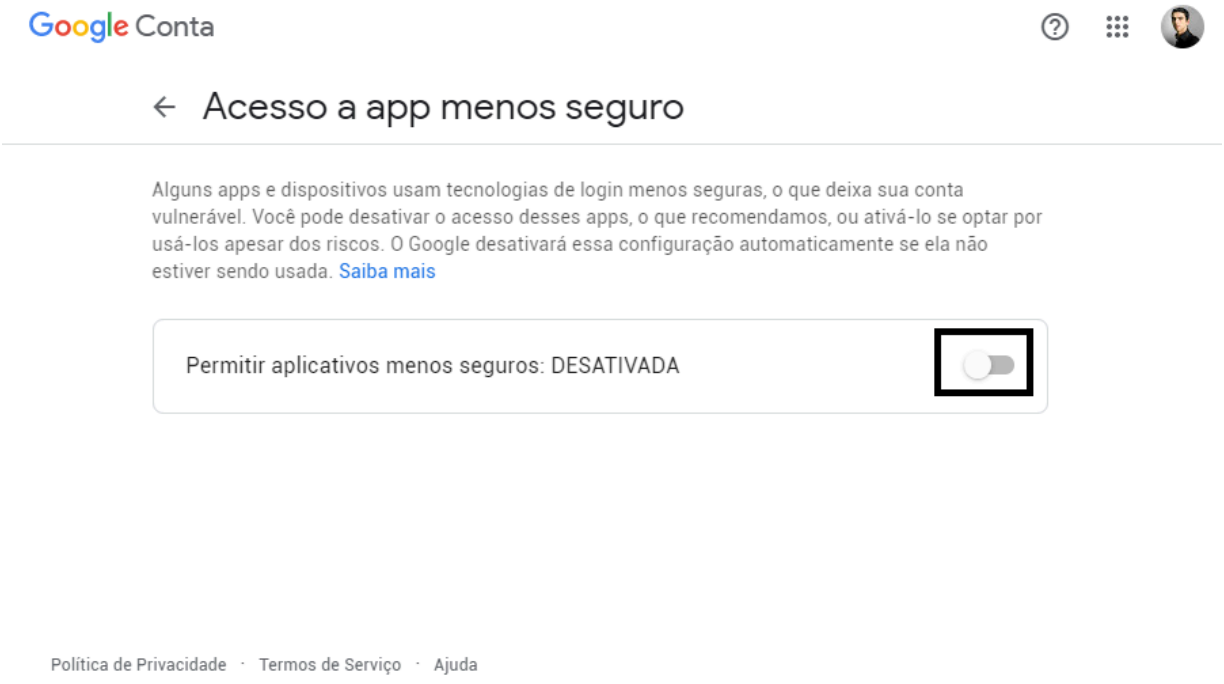


Figura 13.2: Ativar apps menos seguros - Ativado

Clique na seta para voltar:

Figura 13.3: Ativar apps menos seguros - Voltar

Confira se está ativado:

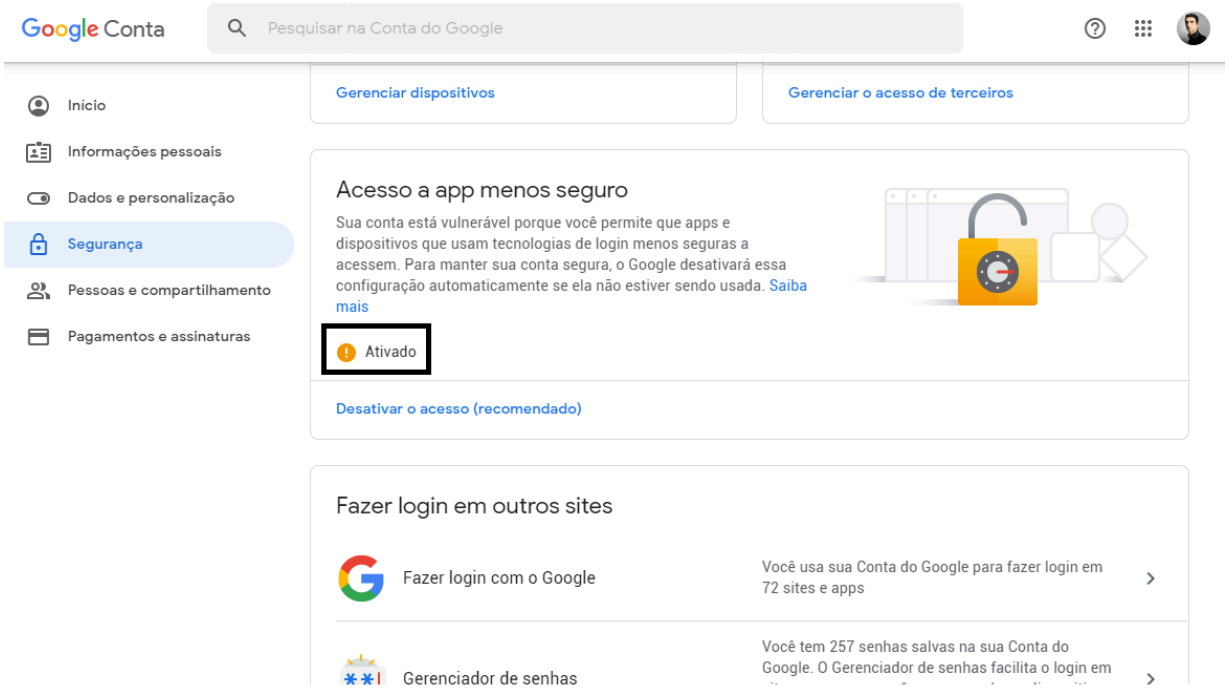


Figura 13.4: Ativar apps menos seguros - Conferir status

Agora podemos voltar ao nosso sistema e configurar o serviço de e-mail.

Arquivo de configuração

Precisamos informar ao Django as credenciais de acesso ao nosso e-mail para que ele possa realizar os envios pela sua própria ferramenta de envio de e-mail. Abra o arquivo `config.py` que fica na pasta `medicSearchAdmin/settings` e adicione as configurações a seguir:

medicSearchAdmin/settings/settings.py

```
# Adicione as linhas a seguir ao final do arquivo
EMAIL_USE_TLS = True
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'seuemail@gmail.com'
EMAIL_HOST_PASSWORD = 'sua senha'
EMAIL_PORT = 587
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
EMAIL_USE_SSL = False
```

Vamos entender o que cada atributo representa:

- **EMAIL_USE_TLS**: campo que diz se será usada a criptografia TLS ou não no envio. Verifique com o provedor do e-mail e caso tenha suporte a TLS deixe `True` .
- **EMAIL_HOST**: host do seu provedor de e-mail, geralmente também é informado na documentação do provedor de seu e-mail.
- **EMAIL_HOST_USER**: e-mail que será usado para fazer o envio dos e-mails.
- **EMAIL_HOST_PASSWORD**: senha do e-mail que será usado.
- **EMAIL_PORT**: porta que o provedor usará para fazer o envio dos e-mails, geralmente você consegue na documentação do provedor.
- **DEFAULT_FROM_EMAIL**: e-mail padrão para servir de remetente quando não colocarmos um remetente em nossos e-mails.

- **EMAIL_USE_SSL**: campo que informa se será usada a criptografia SSL no envio. Geralmente também é informado na documentação do provedor de e-mail.

> Essas informações geralmente ficam na documentação do provedor, mas, caso não esteja, entre em contato com o suporte de sua hospedagem que eles informarão tudo o que você precisa saber.

Com as nossas configurações feitas, precisamos modificar nossa `view` para realizar efetivamente o envio de um e-mail. Abra o arquivo `AuthView.py` que fica na pasta `medicSearch/views` e vamos começar a modificá-lo:

medicSearch/views/AuthView.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.models import User
from medicSearch.models.Profile import Profile
from medicSearch.forms.AuthForm import LoginForm, RegisterForm,
RecoveryForm
# Import as linhas a seguir
from django.conf import settings
from django.core.mail import send_mail
from django.template.loader import render_to_string
from django.utils.html import strip_tags
import hashlib
# Até aqui

... código oculto não apague

def send_email(profile):
    # Altere o método send_email
    profile.token = hashlib.sha256().hexdigest()
    profile.save()
    try:
```

```
        html_message = render_to_string('auth/recovery.html', {'token':
profile.token})
        message = strip_tags(html_message)
        send_mail(
            subject="Recuperação de senha", message=message,
html_message=html_message,
            from_email=settings.EMAIL_HOST_USER, recipient_list=
[profile.user.email], fail_silently=False,
        )
    except:
        raise Exception
```

Vamos entender o que foi feito:

- **from django.conf import settings**: módulo que permite que acessemos informações do arquivo `settings.py`. Aqui usado para pegar o endereço de e-mail que está no `settings` para servir de remetente.
- **from django.core.mail import send_mail**: esse é o método principal do módulo de envio que realiza a junção do assunto, remetente, destinatário e corpo do e-mail.
- **from django.template.loader import render_to_string**: renderiza um template html já no formato string, que é o necessário para o envio ser realizado. Ele converte o html em string, preservando suas tags para o envio ocorrer e chegar ao destinatário com a formatação html.
- **from django.utils.html import strip_tags**: esse módulo formata o html da maneira certa para que seja enviado por e-mail, removendo espaços desnecessários e tags que não são aceitas no envio do e-mail, desse modo, o corpo do texto fica mais leve para ser enviado, tendo somente o necessário.
- **import hashlib**: módulo que usaremos para gerar uma hash 256 para criar o token que será usado pelo usuário para recuperar a senha.

Perceba que usamos um template para enviar a mensagem. Ele funciona exatamente igual ao `render template`. Passamos o nome do arquivo, uma vírgula e um dicionário de contexto com informações

dinâmicas, desse modo podemos ter o campo de token dinâmico no template. Já vamos criá-lo.

Se você não quiser enviar no formato html, mas sim usando um texto plano é só modificar para algo assim:

```
def send_email(profile):
    profile.token = hashlib.sha256().hexdigest()
    profile.save()
    try:
        message = "Olá querido usuário. Clique nesta url para recuperar
sua senha. Url de recuperação: http://localhost:8000/change-
password/{0}".format(profile.token)
        send_mail(
            subject="Recuperação de senha", message=message,
            from_email=settings.EMAIL_HOST_USER, recipient_list=
[profile.user.email], fail_silently=False,
        )
    except:
        raise Exception
```

A diferença é que podemos personalizar o html. Apesar de o nosso não estar personalizado, usaremos o modelo com html para aprendermos como fazer. Crie dentro da pasta `medicSearch/templates/auth` um arquivo chamado `recovery.html`, onde vamos criar nosso template de envio de e-mail:

medicSearch/templates/auth/recovery.html

```
<h3 style="text-transform: uppercase">Recuperação de senha</h3>
<br>
Olá querido usuário. Clique nesta url para recuperar sua senha. Url de
recuperação: http://localhost:8000/change-password/{{ token }}
<br>
<small>Atenciosamente, equipe de Suporte</small>
```

Perceba que passamos como parâmetros do método `render_to_string` o caminho do template `html` e um dicionário com uma chave chamada `token`, que tem como valor `profile.token`.

Poderíamos passar o nome do usuário e qualquer outro dado que quiséssemos renderizar em nosso html. Esse dicionário funciona da mesma maneira que o dicionário de contexto que passamos para a função `render` quando queremos renderizar uma tela `html` no navegador.

> Não adianta passar arquivos `css`, por exemplo. Se quiser personalizar o html, terá que fazer usando o que é chamado de `css inline`, igual ao que fiz no `h3` quando coloquei `<h3 style="text-transform: uppercase">`.

Com isso, receberemos um e-mail de recuperação. Veja o fluxo que já está funcionando.

- Tela de recuperação:

Figura 13.5: Tela de recuperação

- Solicitação de recuperação de senha:

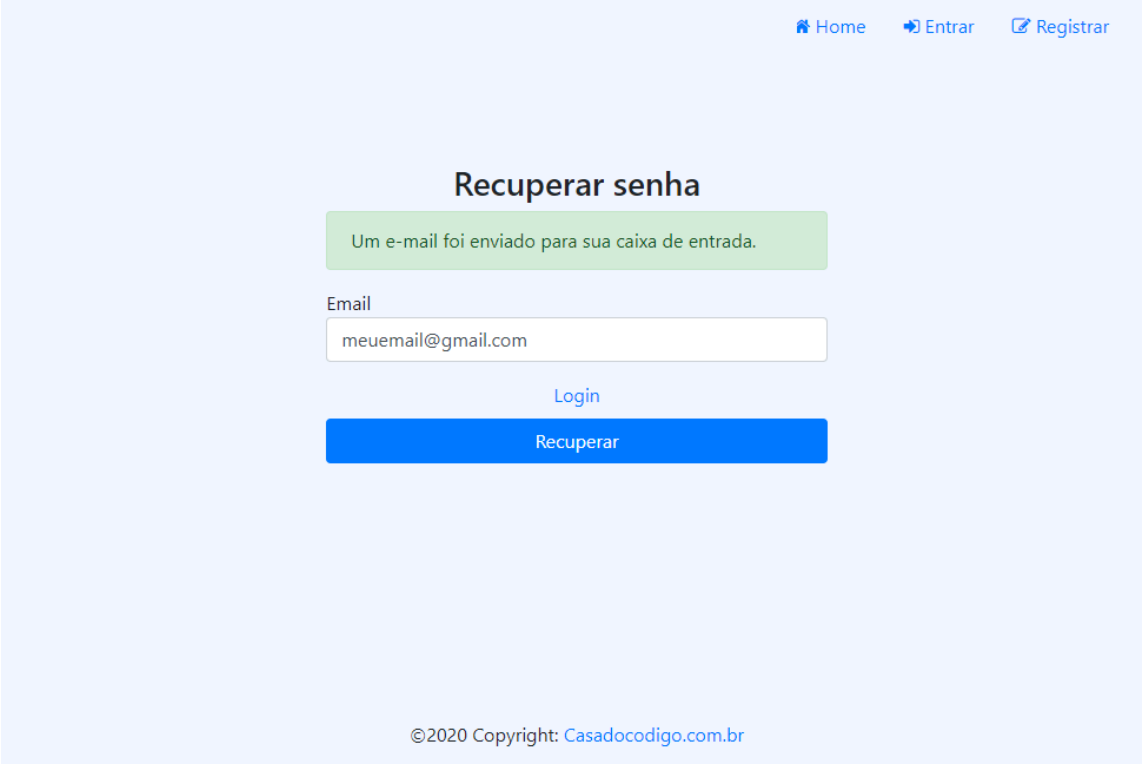


Figura 13.6: Solicitação de recuperação de senha

- E-mail:

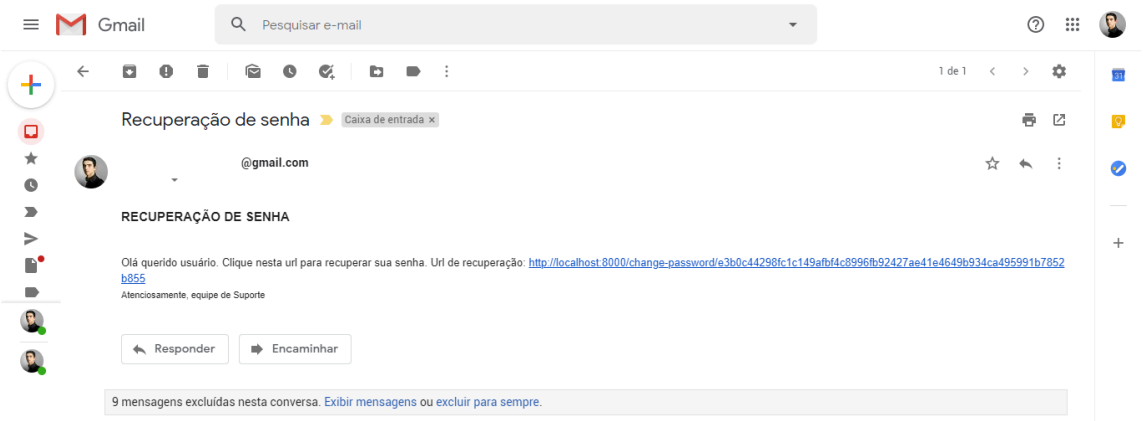


Figura 13.7: E-mail

O fluxo de envio está funcionando! Mas ainda temos um problema. Se clicarmos no link ainda não existe uma tela para alteração de

senha. Então vamos criar o fluxo que vai alterar nossa senha. Abra o arquivo `AuthForm.py`, que fica na pasta `medicSearch/forms`, e vamos criar um `form` para isso:

medicSearch/forms/AuthForm.py:

```
# Adicione essa classe ao final do arquivo
class ChangePasswordForm(forms.Form):
    password = forms.CharField(widget=forms.PasswordInput(attrs={'class':
'form-control'}))
```

Agora vamos criar uma `view` para a alteração de senha. Essa `view` vai receber o token que estará na url e verificará se o token é válido; caso seja, ele abrirá o template, permitindo que o usuário troque a senha. Quando o usuário submeter a nova senha, verificaremos novamente o token e o anularemos para que não possa ser usado de novo.

Mão na massa. Abra o arquivo `AuthView.py` que fica na pasta `medicSearch/views` e vamos começar:

medicSearch/views/AuthView.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.models import User
from medicSearch.models.Profile import Profile
# Import a classe ChangePasswordForm a seguir
from medicSearch.forms.AuthForm import LoginForm, RegisterForm,
RecoveryForm, ChangePasswordForm
from django.conf import settings

...código oculto não apague

# Adicione esse método ao final do arquivo
def change_password(request, token):
    profile = Profile.objects.filter(token=token).first()
    changePasswordForm = ChangePasswordForm()
    message = None
    link_text = 'Solicitar recuperação de senha'
```

```
    link_href = '/recovery'

    if profile is not None:
        if request.method == 'POST':
            changePasswordForm = ChangePasswordForm(request.POST)
            if changePasswordForm.is_valid():
                profile.user.set_password(request.POST['password'])
                profile.token = None
                profile.user.save()
                profile.save()
                message = { 'type': 'success', 'text': 'Senha alterada com
sucesso!!!' }
                link_text = 'Login'
                link_href = '/login'
            else:
                message = { 'type': 'danger', 'text': 'Formulário
inválido.' }
    else:
        message = { 'type': 'danger', 'text': 'Token inválido. Solicite
novamente.' }

    context = {
        'form': changePasswordForm,
        'message': message,
        'title': 'Alterar senha',
        'button_text': 'Alterar',
        'link_text': link_text,
        'link_href': link_href
    }
    return render(request, template_name='auth/auth.html',
context=context, status=200)
```

Esse método basicamente vai verificar se o token passado é válido; caso seja, ele permitirá que o usuário altere a senha; caso não seja, ele já informará que o token não é válido e que não será possível trocar de senha.

Com isso só precisamos criar nossa url de recuperação. Abra o arquivo `AuthUrls.py` que fica na pasta `medicSearch/urls` e vamos

adicionar a url de alteração de senha:

medicSearch/urls/AuthUrls.py

```
from django.urls import path, include
# Import o método change_password
from medicSearch.views.AuthView import login_view, register_view,
logout_view, recovery_view, change_password

urlpatterns = [
    path("login", login_view, name='login'),
    path("register", register_view, name='register'),
    path("logout", logout_view, name='logout'),
    path("recovery", recovery_view, name='recovery'),
    # Adicione a linha a seguir
    path("change-password/<str:token>", change_password, name='change-
password'),
    # Até aqui
    path('social-auth/', include('social_django.urls',
namespace="social")),
]
```

Pronto, com isso nosso fluxo está completo e já poderemos clicar no link do e-mail e alterar a senha. Veja a continuação do fluxo a seguir:

Após clicar no link do e-mail:

Figura 13.8: Após clicar no link do e-mail

Solicitando a troca da senha:

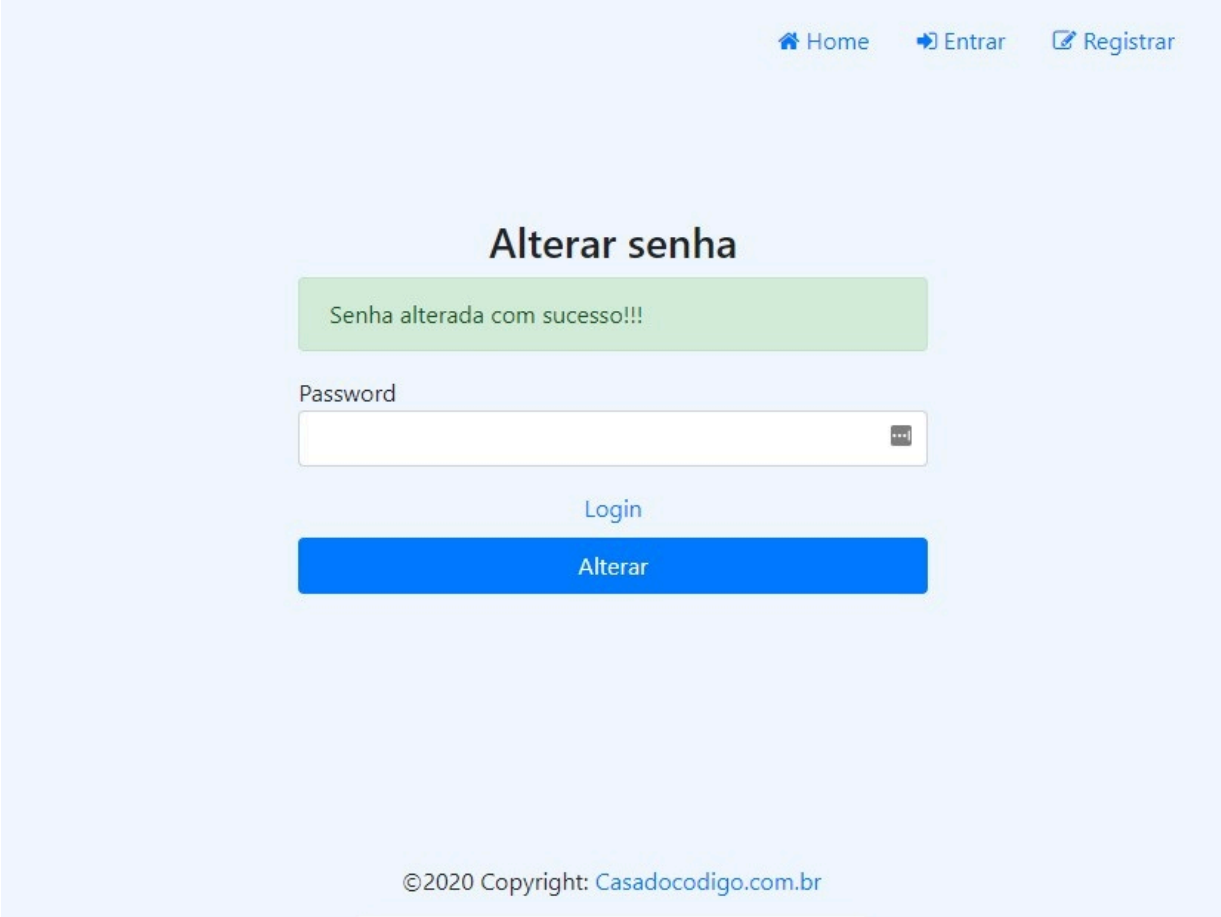


Figura 13.9: Solicitando a troca da senha

Se você tentar realizar o login com o usuário já vai conseguir com sucesso! Agora nosso serviço de e-mail está concluído. Você poderá implementá-lo em outras partes do sistema, como confirmação de e-mail no cadastro, notificação de atualizações para os usuários e muitas outras coisas.

Capítulo 14
Testes unitários

14.1 Introdução

Neste capítulo, aprenderemos o que são testes unitários. O principal objetivo aqui é demonstrar como aplicar um teste unitário através das próprias ferramentas que o Django nos proporciona, demonstrando as grandes facilidades que existem ao utilizar sua classe principal, a `TestCase`. Vamos lá!

14.2 O que são testes unitários

Testes unitários ou testes de unidade são basicamente testes da menor parte testável da aplicação. Se você está desenvolvendo um programa utilizando paradigma orientado a objetos, a menor parte testável do seu programa são os métodos e atributos de uma classe.

Tendo ainda como exemplo um sistema com POO, criaremos testes unitários para testarmos os métodos e atributos de nossa classe. Se nosso programa utilizasse paradigma funcional, a menor parte do programa seriam as funções, então criaríamos testes unitários para testarmos as funções do programa.

Antes de seguir, precisamos entender que não existem apenas testes unitários em uma plataforma, mas existem motivos que veremos a seguir que fazem com eles sejam a principal escolha de um projeto. Veja a seguir outros tipos de testes que existem:

- Teste de instalação e configuração;
- Teste de integridade;
- Teste de segurança;

- Teste funcional;
- Teste de integração;
- Teste de volume;
- Teste de performance;
- Teste de usabilidade;
- Teste de regressão.

Nem todos os testes da lista são automatizados pelo próprio software e por isso podem gerar um alto custo para o projeto.

Por ser um tipo de teste que foca na aplicação de forma particionada, onde cada teste geralmente testa apenas um pedaço do código - seja um método ou atributo, no caso da programação OO, ou uma função, no caso da funcional -, ele se torna um dos tipos de testes mais baratos que existem e por isso é um dos mais implementados entre os citados na listagem anterior. A seguir temos uma imagem da pirâmide de testes.

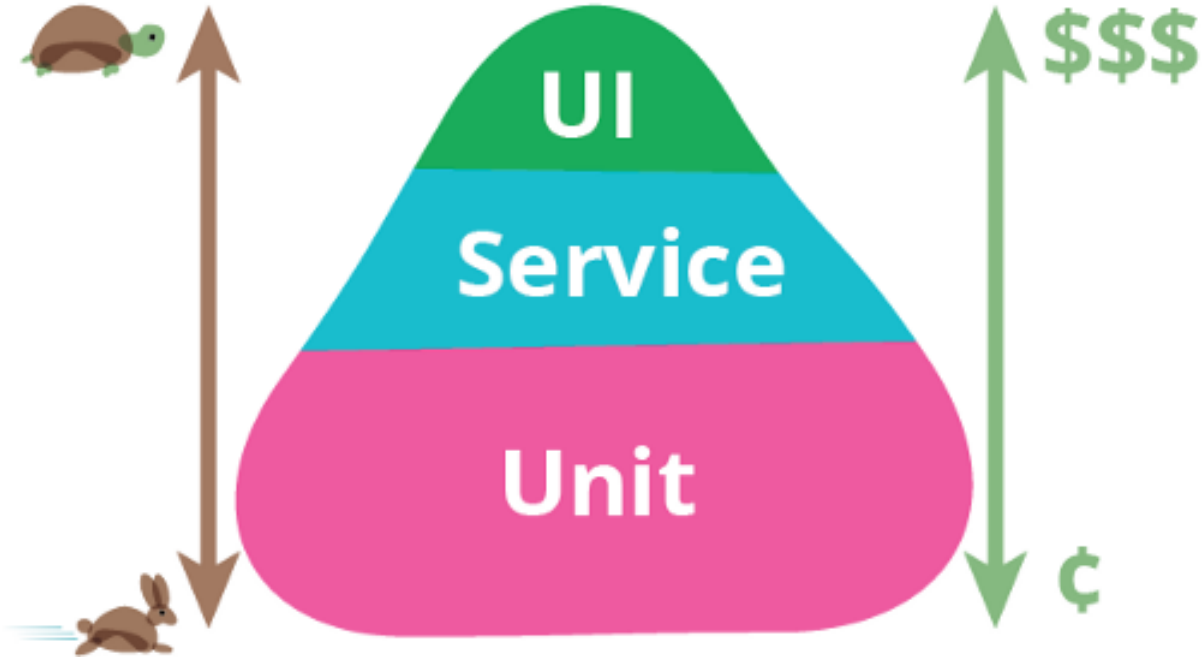


Figura 14.1: Pirâmide de testes

Como podemos perceber, com testes unitários podemos otimizar uma grande quantidade de possíveis erros sem gerarmos um custo

exorbitante para o desenvolvimento do projeto.

Sabemos que os testes de unidade precisam ser desenvolvidos pelo programador e isso vai consumir um pouco mais de tempo dele no que se refere a desenvolvimento do que caso ele não implementasse o teste. Mas no escopo global do desenvolvimento, perceberemos que o custo será menor, pois testes de unidade evitam que um programa suba para o ambiente de teste com alguns erros comuns, já que toda vez que você for lançar uma versão você testará o projeto.

Perceba que você desenvolverá o teste uma única vez e, caso um erro ocorra, você o detectará antes de subir a plataforma para homologação ou produção, desse modo será feita uma correção preventiva no programa, gerando uma grande economia de tempo e que, em longo prazo, trará grandes economias ao projeto.

Veja a seguir um gráfico que explica melhor sobre a curva de defeitos de software e perceba como os testes de unidade são importantes para otimizá-la:

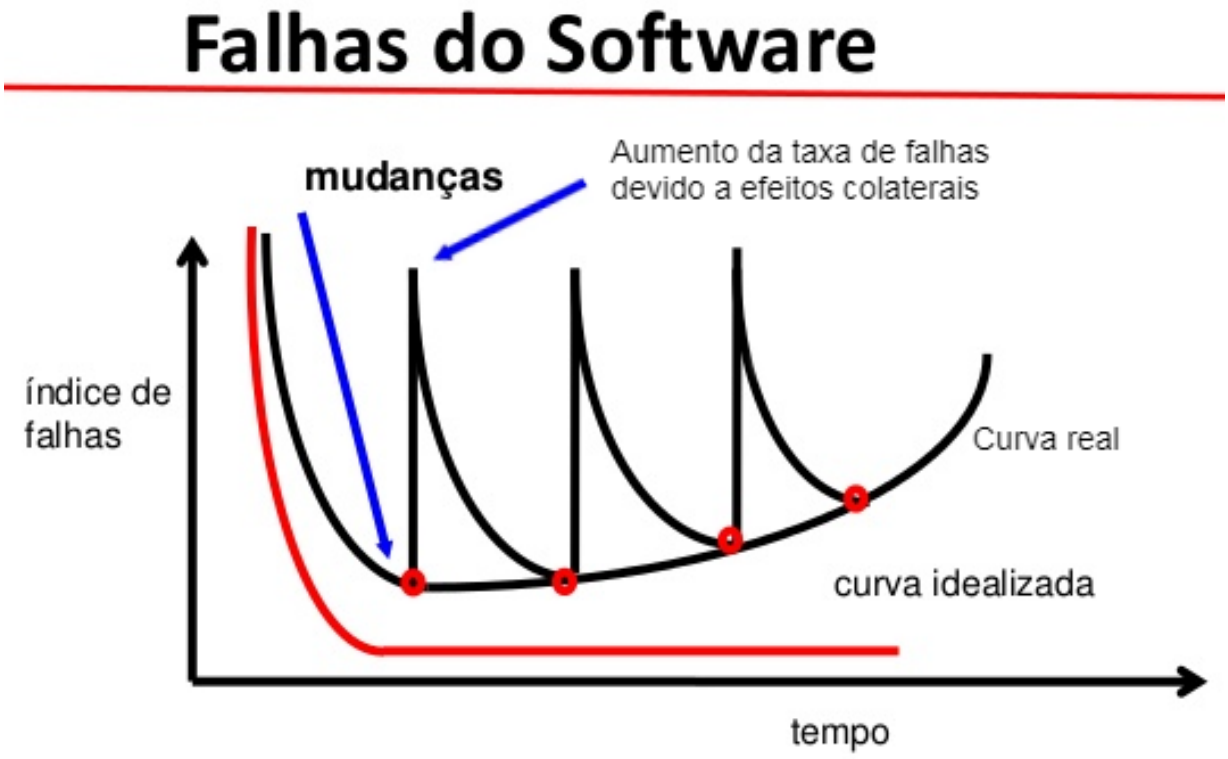


Figura 14.2: Curva de defeitos para softwares de Roger Pressman

O gráfico anterior expressa uma realidade do desenvolvimento de software muito comum, um constante aumento da taxa de falhas devido a efeitos colaterais. Esses efeitos são geralmente aquelas correções feitas em um método/função que, sem você perceber, afetam outra parte do código. Mas se você tiver um teste de unidade daquela parte do código, ao fazer a alteração no método e realizar todo o teste da aplicação, o teste vai acusar que um erro ocorreu.

Você deve estar se perguntando: mas como assim erro? Eu apenas evoluí o código, modifiquei, melhorei ou qualquer coisa do tipo. Os testes de unidade, como quaisquer outros testes automatizados, não têm como função testar se o código está funcionando, mas sim verificar se, após alguma modificação naquele pedaço do código, a sua aplicação continua trazendo o resultado esperado naquele pedaço de código.

Um exemplo tosco, mas que pode refletir como queremos aqui, é: imagine que seu método retorne um valor numérico; sabemos que todos os lugares que implementam a classe que possui esse método provavelmente precisam receber um valor numérico. Um belo dia você decide fazer com que aquele método passe a retornar uma lista, com a posição 0 sendo o valor numérico e a posição 1 sendo uma string. Sem dúvidas, a mudança ocorreu no método, mas todo o código continuará esperando um valor numérico. Quando você rodar o teste, ele vai apontar para você que é necessário mudar a forma de receber o método, pois antes você recebia um número e agora precisa receber uma lista. Isso será detectado antes de você fazer um deploy, desse modo, você conseguirá reduzir o aumento na curva de falhas do seu software.

É claro que os testes de unidade não são apenas isso. Aqui eu trouxe o mais simples dos exemplos que existem, mas, conforme veremos, existem aplicações mais avançadas para o teste de unidade. Desde já, saiba que eles representam cerca de 70% dos testes da plataforma, por serem mais baratos.

14.3 TDD

Em programação existe o que chamamos de desenvolvimento guiado por testes ou, em inglês, TDD (*Test-Driven Development*). O TDD é uma técnica de desenvolvimento que visa começar o desenvolvimento de cada funcionalidade do software pelo seu teste, e é aqui que o teste de unidade entra. Parece ser algo muito diferente, mas não é. Começar a desenvolver uma funcionalidade pelo teste facilita o desenvolvimento, porque desse modo faremos a funcionalidade com o objetivo de passar no teste que foi desenvolvido. Existem três etapas no TDD, veja cada uma delas na imagem a seguir:

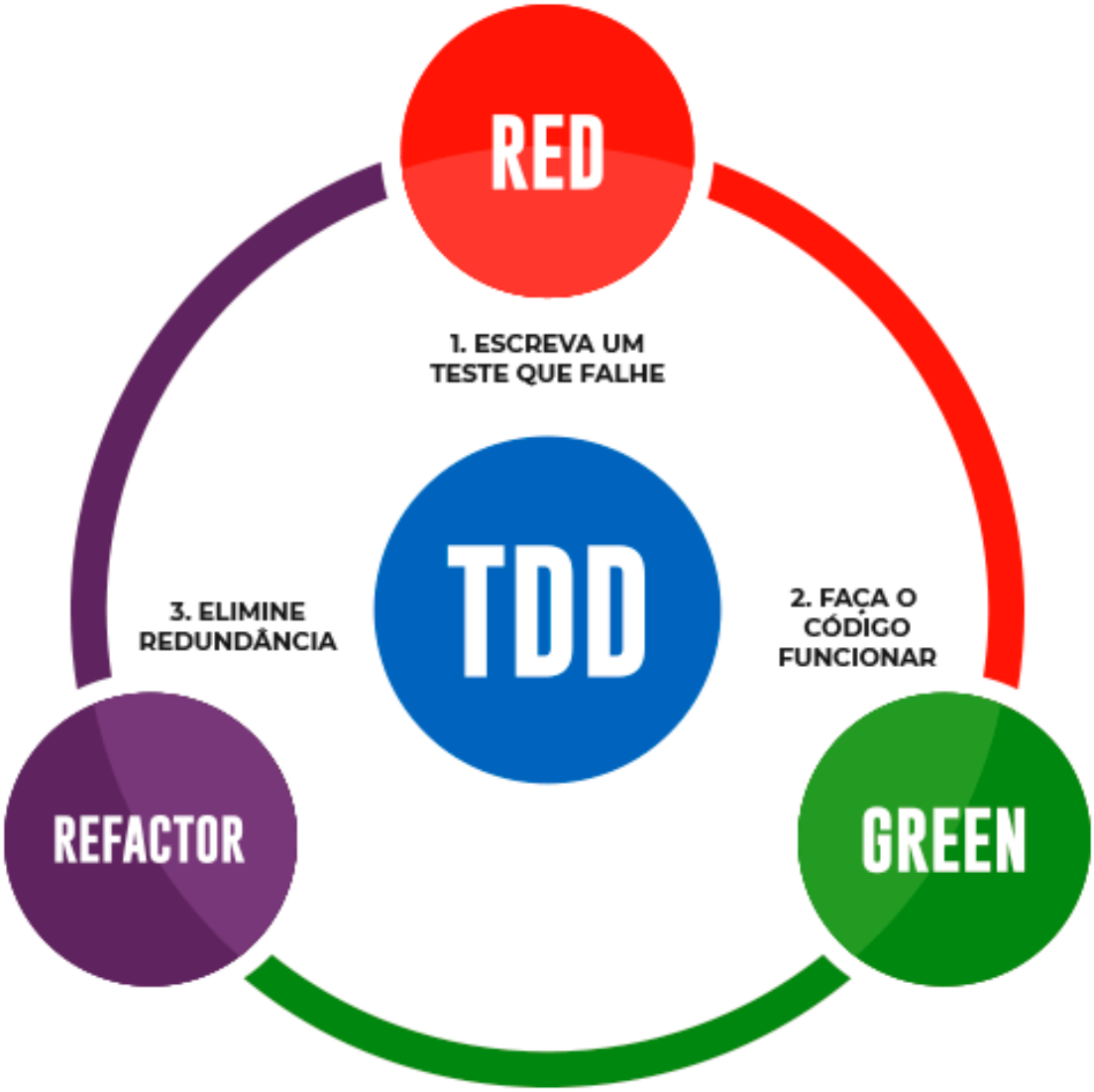


Figura 14.3: As três etapas do TDD

1. **Red (Vermelho)**: escreva um pequeno teste automatizado que, ao ser executado, falhará porque ainda não temos o código da funcionalidade criado.
2. **Green (Verde)**: implemente um código que seja suficiente para ser aprovado no teste recém-escrito. Perceba que aqui você terá o teste já criado na etapa Red para utilizar como base no desenvolvimento da sua funcionalidade.

3. **Refactor (Refatorar)**: refatore o código a fim de ele ser melhorado, deixando-o mais funcional e mais limpo.

Vendo a aplicação das três etapas do TDD, temos em conjunto com ele o modelo *F.I.R.S.T.* que dará uma ótima orientação de quais itens devem ser priorizados na hora de criar e executar os testes:

- **F (Fast)**: rápidos;
- **I (Isolated)**: isolados;
- **R (Repeatable)**: reproduzíveis;
- **S (Self-verifying)**: autoverificáveis;
- **T (Timely)**: oportunos, no momento certo.

Com esse modelo, conseguimos analisar com mais facilidade em quais cenários da nossa aplicação vamos trabalhar com a programação guiada por testes. Não existe uma regra que diga e exija que você tenha todo o seu código sendo desenvolvido em conjunto com testes. Utilize o modelo F.I.R.S.T. e analise quais os melhores lugares para aplicar as três etapas do TDD dentro do seu programa.

Resumindo, entre as principais vantagens do TDD está a de termos um código mais limpo e fácil de passar por uma refatoração.

14.4 Testes unitários no Django

Nesta etapa, veremos alguns recursos do Django que nos permitem desenvolver testes de unidade facilmente através do módulo `django.test`, que já vem instalado no Django e implementa o módulo `unittest`, que é nativo do Python. Veremos também alguns módulos de processamento por thread, requisições HTTPS por background e outros que facilitam bastante o desenvolvimento dos testes unitários. Vamos lá:

Dentro de todo app que criamos no Django já vem um arquivo chamado `tests.py`. Vamos apagá-lo, pois criaremos nossos testes de uma maneira mais organizada. Abra a pasta `medicSearch` e delete o arquivo `tests.py`. Agora crie dentro da pasta `medicSearch` um diretório chamado `tests` e dentro crie um arquivo chamado `__init__.py`. Com essa pasta, o diretório ficará parecido com este a seguir:

```
livro_django
├──medicSearch
│        migrations
│        models
│        static
│        templates
│        tests
│              __init__.py
│        urls
│        views
│        __init__.py
│        ... Arquivos ocultos para otimizar a página
├──medicSearchAdmin
│        ... Arquivos ocultos para otimizar a página
```

14.5 Primeiro teste unitário

Vamos criar nosso primeiro teste de unidade no projeto, que listará um usuário do Django e, caso retorne um valor, o teste estará correto. Abra o diretório `medicSearch/tests` e crie dentro dele um arquivo chamado `test_model_user.py`. Vamos adicionar o código a seguir:

medicSearch/tests/test_model_user.py

```
from django.test import TestCase
from django.contrib.auth.models import User
```

```
class UserModelTestClass(TestCase):
    def setUp(self):
        pass

    def test_user_exist(self):
        user = User.objects.first()
        self.assertIsNotNone(user)
```

Perceba que criamos a classe `UserModelTestClass` que herda de `TestCase`. `TestCase` possui diversos módulos que podemos utilizar nos testes de unidade para verificar as saídas esperadas em nossos testes. Entre eles existe o principal, o `setUp`, que veremos mais a seguir.

Realizamos a listagem de um usuário e após isso verificamos através do método `assertIsNotNone` se o resultado trazido é nulo ou não. Caso retorne algo, o teste vai passar; caso retorne um resultado nulo, o teste de unidade vai reclamar. Rode o comando `python manage.py test` em seu console e veja que receberá uma saída similar à seguinte:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py test
Creating test database for alias 'default'...
AssertionError: unexpectedly None
----------------------------------------------------------------------
Ran 1 test in 0.002s
FAILED (failures=1)
Destroying test database for alias 'default'...
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Esse teste não foi bem-sucedido e você deve estar se perguntando o motivo. Os testes unitários não utilizam e nem poderiam utilizar a mesma base de dados que utilizamos em nossa aplicação de desenvolvimento. Por isso, precisamos configurar um `Setup` inicial para cada teste unitário que criamos. Vamos ver a seguir como realizar isso. Altere o arquivo `test_model_user.py` para ficar igual ao seguinte:

medicSearch/tests/test_model_user.py

```
from django.test import TestCase
from django.contrib.auth.models import User

class UserModelTestClass(TestCase):
    def setUp(self):
        User.objects.create(username='teste.unitario', password='123456')

    def test_user_exist(self):
        user = User.objects.first()
        self.assertIsNotNone(user)
```

Como falamos anteriormente, os testes de unidade não olham para o banco de desenvolvimento. Perceba que quando rodamos o comando `python manage.py test` a primeira linha que roda é `Creating test database for alias 'default'...` e a última é `Destroying test database for alias 'default'...` , ou seja, ao iniciar o teste, ele cria um banco para realizar os testes e depois o destrói. Desse modo, antes de realizar os testes, precisamos popular esse banco com os dados de que precisamos para realizar os testes necessários. Não é nada complicado, você não precisa abrir conexão nem nada do tipo, apenas adicione os dados necessários através do método `setUp` que é herdado da classe `TestCase` .

Se rodarmos o comando `python manage.py test` novamente, teremos uma saída diferente agora. Veja:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.003s
OK
Destroying test database for alias 'default'...
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Já que criamos um usuário no método `setUp` , agora, quando chamamos o método `test_user_exist` , o retorno dele será o usuário

criado no `setUp` e, ao verificar o retorno com o método `assertIsNotNone`, ele retornará o valor `True` e o teste passará.

Existem outros métodos do módulo `unittest` que podemos utilizar para saber se um determinado resultado está retornando o que é esperado. A seguir temos uma lista com os principais:

- **assertFalse(self, expr, msg=None)**: verifica se o valor do `return` é falso.
- **assertTrue(self, expr, msg=None)**: verifica se o valor do `return` é verdadeiro.
- **assertEqual(self, first, second, msg=None)**: verifica se o parâmetro `first` é igual ao parâmetro `second`.
- **assertNotEqual(self, first, second, msg=None)**: verifica se o parâmetro `first` é diferente do parâmetro `second`.
- **assertListEqual(self, list1, list2, msg=None)**: verifica se a lista do parâmetro `list1` é igual à do `list2`.
- **assertTupleEqual(self, tuple1, tuple2, msg=None)**: verifica se a tupla do parâmetro `tuple1` é igual à do `tuple2`.
- **assertSetEqual(self, set1, set2, msg=None)**: verifica se o conjunto do parâmetro `set1` é igual ao do `set2`.
- **assertDictEqual(self, d1, d2, msg=None)**: verifica se o dicionário do parâmetro `d1` é igual ao do `d2`.
- **assertCountEqual(self, first, second, msg=None)**: verifica entre dois elementos iteráveis se possuem a mesma quantidade de elementos.
- **assertIsNone(self, obj, msg=None)**: verifica se o resultado é nulo.
- **assertIsNotNone(self, obj, msg=None)**: verifica se o resultado não é nulo.
- **assertContains(self, response, text)**: verifica se no parâmetro `response` contém a string que é passada no parâmetro `text`.

Existem outros métodos, caso deseje conhecê-los, consulte a documentação do `python unittest` no tópico sobre os métodos assets (https://docs.python.org/3/library/unittest.html#assert-methods).

14.6 Usando o Client para fazer requisições

Agora, veremos a classe `Client` do módulo `django.test`. Com ela, é possível realizarmos requisições HTTPS dentro dos nossos testes de unidades. Vamos criar a seguir um pequeno exemplo de `get` que retornará para nós o resultado `html` da `view` de médicos. Depois, veremos se o status de resposta é igual a 200 e se existe um determinado texto dentro do `html`.

Crie dentro da pasta `medicSearch/tests` um arquivo chamado `test_medic_view.py` e adicione o código a seguir:

medicSearch/tests/test_medic_view.py

```
from django.test import TestCase, Client
from django.contrib.auth.models import User
from medicSearch.models.Profile import Profile
from django.db import transaction, IntegrityError

class MedicViewTestClass(TestCase):
    def setUp(self):
        try:
            with transaction.atomic():
                user = User.objects.create(username='teste.unitario',
password='123456')
                profile = Profile.objects.get(user=user)
                profile.role = 2
                profile.save()
        except IntegrityError as e:
            print("Erro ao criar usuário. Descrição: %s" % e)

        self.client = Client()

    def test_medics_list(self):
        response = self.client.get('/medic/')
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Foram encontrados: 1 medico(s)')
```

Perceba que agora criamos uma `transaction` em nosso `setUp`. O padrão de um user inicial é ser criado como usuário, então foi necessário alterá-lo para ser um médico, pois precisávamos listar pelo menos um médico em nossa `view`. Além disso, foi preciso instanciar o método `Client`.

Se você tiver algum problema de exigência do `CSRF_TOKEN` no momento do teste, pode passar como parâmetro para a classe `Client` o valor `enforce_csrf_checks=False`. Desse modo, o Django não vai exigir que haja um `token csrf` dentro da requisição. Isso pode ser exigido se você for realizar um teste simulando um `POST` de login, por exemplo.

A classe `Client` possui todos os métodos de requisição HTTP. Veremos a seguir um exemplo real, no qual criaremos um teste com o login da nossa plataforma utilizando o método `POST`. Os demais métodos que diferem são similares ao `GET` que já vimos e o `POST` que veremos a seguir:

> Você deve ter visto que, em vez de utilizar uma consulta como `Profile.objects.filter(user=user).first()`, foi utilizado `Profile.objects.get(user=user)`. Esse é um modo de buscar um único resultado, onde buscamos no banco baseado em um parâmetro. Neste caso, usamos `user`, mas poderia ter sido `Profile.objects.get(pk=1)`, onde `pk` é o `id` do registro no banco de dados. Desse modo, não é preciso utilizar o método `first()` para dizer que você quer retornar apenas um objeto como resultado.

14.7 Criando um teste de login

Para juntar o conhecimento adquirido, vamos criar um teste de login, que terá tudo que vimos até agora dentro deste capítulo. Crie

dentro da pasta `medicSearch/tests` um arquivo chamado `test_login.py` e adicione o código a seguir:

medicSearch/tests/test_login.py

```
from django.test import TestCase, Client
from django.contrib.auth.models import User

class LoginTestClass(TestCase):
    def setUp(self):
        User.objects.create(username='teste.unitario', password='123456')
        self.client = Client()

    def test_login(self):
        response = self.client.post('/login', {
            'username': 'teste.unitario',
            'password': '123456'
        }, **{'Content-Type': 'application/x-www-form-urlencoded'})
        self.assertEqual(response.status_code, 200)
```

Como podemos ver, com tudo que foi falado, é possível criarmos testes de unidade para nosso programa sem dificuldade.

Perceba que no login substituímos o método `get()` pelo `post()` e, além do primeiro parâmetro que passamos do `endpoint`, foi passado também um segundo parâmetro, do tipo dicionário com os valores que devem ser enviados para a API de login e um terceiro, que é o que chamamos de *magic variable* em Python (nesse caso, o famoso `kwargs`, que contém um dicionário em que podemos passar as `headers` da requisição).

> Através da lib `Client` é possível usar todos os métodos que representam os verbos HTTP: `get`, `post`, `put`, `patch`, `delete` etc. Todos podem se comportar da mesma maneira que vimos anteriormente com esses três parâmetros.

14.8 Cobertura de código

Antes de concluir, vamos instalar uma biblioteca que nos permitirá ver o quanto nosso código está coberto por testes de unidade. Isso é legal para termos uma ideia dos testes já feitos e os que ainda não foram criados para a plataforma. É bem útil para projetos que já existem e ainda não possuem testes de unidade e agora você precisa ter uma ideia do quanto o seu código já está coberto após começar a implementar testes de unidade.

Vamos instalar o módulo `coverage` rodando o comando `pip install coverage`:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install coverage
Collecting coverage
  Using cached coverage-5.2-cp38-cp38-win32.whl (206 kB)
Installing collected packages: coverage
Successfully installed coverage-5.2
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Agora que foi instalado, podemos rodar o comando `coverage run manage.py test` sempre que quisermos. Ele rodará nossos testes unitários, gerará um banco de dados `sqlite` com o relatório dos testes e poderemos ver a cobertura do código a qualquer momento. Vamos ver a seguir as duas maneiras de ver a cobertura dos testes.

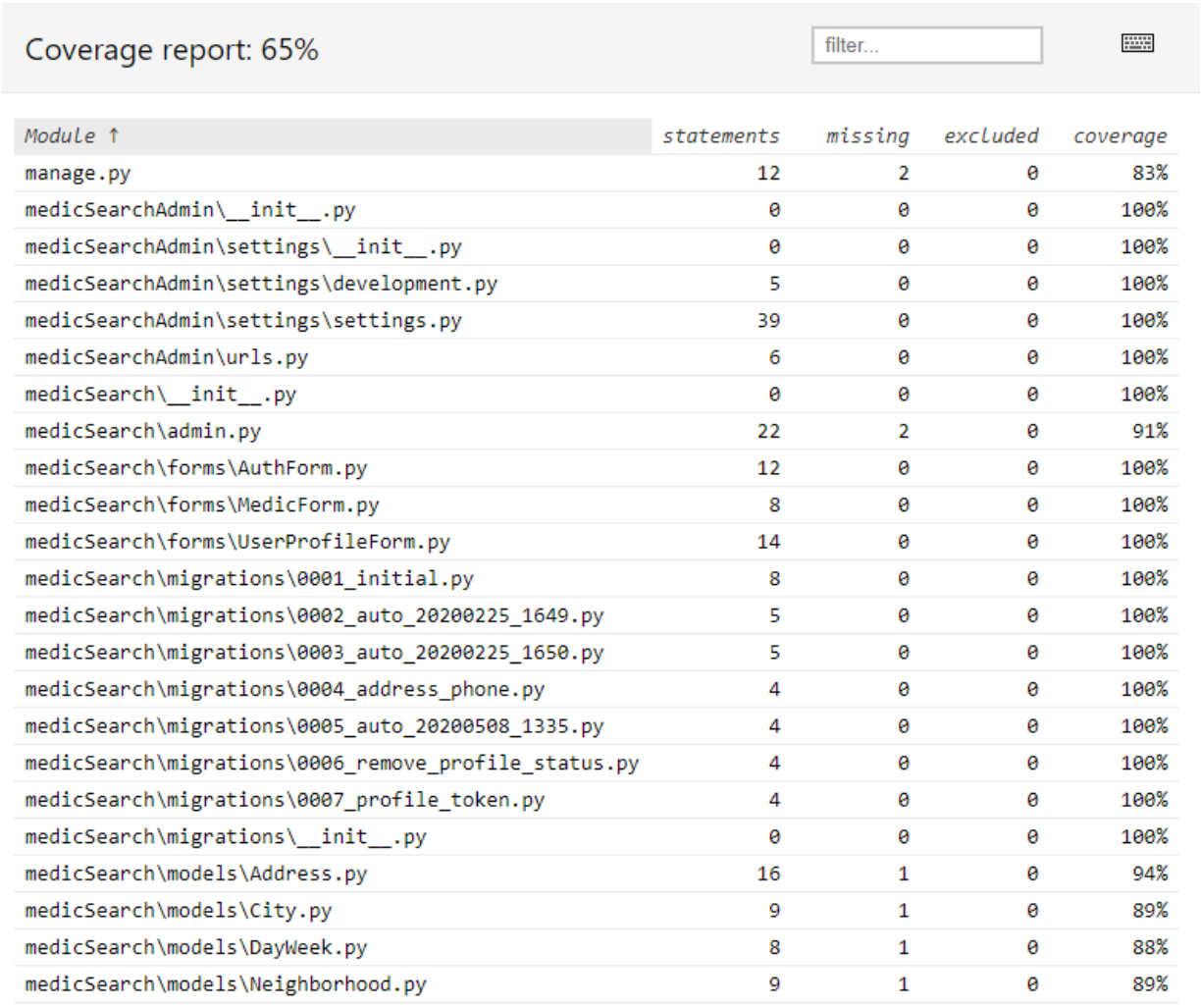
Para ver a cobertura do código pelo terminal, rode `coverage report --omit=*/venv/*` e terá um resultado similar ao do terminal a seguir:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ coverage report
Name                                                                 Stmts   Miss  Cover
----------------------------------------------------------------------------------------
manage.py                                                               12      2    83%
medicSearchAdmin\__init__.py                                             0      0   100%
medicSearchAdmin\settings\__init__.py                                    0      0   100%
medicSearch\views\AuthView.py                                          100     72    28%
medicSearch\views\HomeView.py                                            3      1    67%
medicSearch\views\ProfileView.py                                        42     34    19%
```

```
medicSearch\views\__init__.py                              3      0   100%
-----------------------------------------------------------------------------
TOTAL                                                    578    200    65%

(venv) tiago_luiz@ubuntu:~/livro_django$
```

Eu precisei ocultar o resultado total, porque eram muitas linhas, mas será algo parecido com isso. Você também pode gerar o resultado em um arquivo `HTML` para ver em seu navegador. Rode o comando `coverage html` no terminal e uma pasta chamada `htmlcov` será criada dentro do projeto. Abra o arquivo `index.html` que fica dentro da pasta e você verá uma tela igual à da imagem a seguir:



Coverage report: 65%

filter...

Module ↑	statements	missing	excluded	coverage
manage.py	12	2	0	83%
medicSearchAdmin__init__.py	0	0	0	100%
medicSearchAdmin\settings__init__.py	0	0	0	100%
medicSearchAdmin\settings\development.py	5	0	0	100%
medicSearchAdmin\settings\settings.py	39	0	0	100%
medicSearchAdmin\urls.py	6	0	0	100%
medicSearch__init__.py	0	0	0	100%
medicSearch\admin.py	22	2	0	91%
medicSearch\forms\AuthForm.py	12	0	0	100%
medicSearch\forms\MedicForm.py	8	0	0	100%
medicSearch\forms\UserProfileForm.py	14	0	0	100%
medicSearch\migrations\0001_initial.py	8	0	0	100%
medicSearch\migrations\0002_auto_20200225_1649.py	5	0	0	100%
medicSearch\migrations\0003_auto_20200225_1650.py	5	0	0	100%
medicSearch\migrations\0004_address_phone.py	4	0	0	100%
medicSearch\migrations\0005_auto_20200508_1335.py	4	0	0	100%
medicSearch\migrations\0006_remove_profile_status.py	4	0	0	100%
medicSearch\migrations\0007_profile_token.py	4	0	0	100%
medicSearch\migrations__init__.py	0	0	0	100%
medicSearch\models\Address.py	16	1	0	94%
medicSearch\models\City.py	9	1	0	89%
medicSearch\models\DayWeek.py	8	1	0	88%
medicSearch\models\Neighborhood.py	9	1	0	89%

Figura 14.4: HTML com porcentagem de cobertura de testes

Ao clicar em qualquer link, você verá a descrição das áreas não cobertas do código.

Figura 14.5: HTML com porcentagem de cobertura de testes

Nessa última imagem, vemos a parte do teste de login que nosso teste não cobriu. Com isso, vemos como é possível ter controle da cobertura de testes que nosso código possui.

Acostumar-se a trabalhar desenvolvendo testes antes das funcionalidades pode ser algo complicado no início, mas com o passar do tempo se tornará costume. A maioria das empresas tem exigido programadores que desenvolvam e tenham conhecimento em TDD e dentro dele principalmente os testes de unidade. Sem dúvidas, é válido aprender a trabalhar com os demais testes que

existem em desenvolvimento de softwares, mas o teste de unidade é, além de um ótimo começo, uma grande *skill* que todo desenvolvedor precisa ter para se destacar no mercado.

Capítulo 15
Deploy no Heroku

15.1 Introdução

Neste capítulo, concluiremos nosso projeto realizando o deploy em uma plataforma chamada Heroku, atualmente uma das mais utilizadas no mundo.

Antes de começar as instalações, vamos entender o que o Heroku fornecerá para nós. Heroku é uma plataforma em nuvem como um serviço que suporta várias linguagens de programação. Foi uma das primeiras plataformas em nuvem a surgir, estando em desenvolvimento desde junho de 2007, quando suportava apenas a linguagem de programação Ruby, mas agora já suporta Java, Node.js, Scala, Clojure, Python, PHP e Go. (Definição e contextualização retiradas da Wikipédia. Fonte: https://en.wikipedia.org/wiki/Heroku).

Para realizar o deploy, o Heroku utiliza o sistema de versionamento GIT. A plataforma ficará dentro de um contêiner (*container*) que o Heroku criará para o projeto. Não falaremos sobre contêineres no livro, mas basicamente uma ótima explicação e simples sobre um contêiner é:

> Cada recipiente mantém os componentes necessários para executar o software desejado, como banco de dados, variáveis de ambientes e bibliotecas. Com isso, um único contêiner não é capaz de consumir toda a memória e CPU do host disponível, ou seja, sua aplicação ficará em um contêiner separado de outras aplicações que você possua dentro do Heroku.

15.2 Criando uma conta no Heroku

Existem planos pagos, mas também há um plano gratuito, que poderemos usar para fazer o deploy da plataforma. Vamos criar nossa conta no Heroku através da url https://signup.heroku.com/. Após preencher os dados, você receberá um e-mail com um link para confirmar sua conta.

Ao clicar no link você será direcionado para uma tela onde terá que criar sua senha.

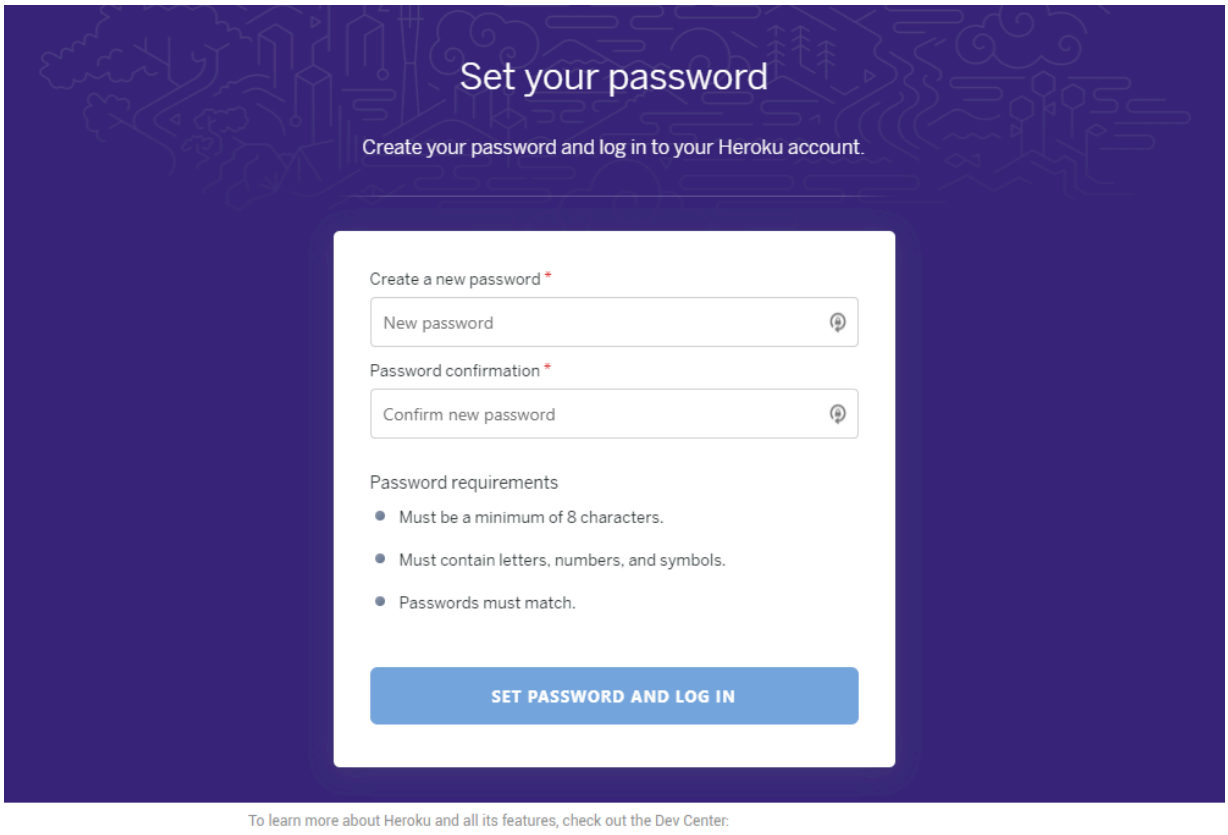


Figura 15.1: Criar senha da conta Heroku

Com isso, você já tem uma conta do Heroku para realizar deploy. Agora precisamos instalar o `Heroku CLI` para podermos rodar os comandos que realizarão o deploy da aplicação.

15.3 Instalando o Heroku CLI

MacOS

Para instalar o `Heroku CLI` no macOS, rode o comando `brew tap heroku/brew && brew install heroku` em seu terminal:

```
(venv) TiagoSilva@MacBook-Air:~/livro_django$ brew tap heroku/brew && brew install heroku
```

Ubuntu 16 ou superior

Para instalar o `Heroku CLI` no macOS, rode o comando `sudo snap install --classic heroku` em seu terminal:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ sudo snap install --classic heroku
```

Via Curl

O comando `curl https://cli-assets.heroku.com/install.sh | sh` poderá ser utilizado para instalar em outros sistemas baseados em Linux que tenham o `Curl` instalado:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ curl https://cli-assets.heroku.com/install.sh | sh
```

Windows

Para instalar no Windows, você precisará baixar o executável para 64 ou 32 bits:

- **64-bit**: https://cli-assets.heroku.com/heroku-x64.exe
- **32-bit**: https://cli-assets.heroku.com/heroku-x86.exe

Após baixar e instalar no Windows, feche o terminal que estiver usando e abra-o novamente. Terminais já abertos não atualizam as variáveis de ambiente, então você tentará rodar os comandos do Heroku e ele não reconhecerá.

Após instalar de qualquer uma dessas maneiras, você poderá rodar comandos do Heroku em seu computador. Em breve vamos usar esses comandos para realizar o deploy da aplicação.

15.4 Preparando o projeto

Para que nosso projeto rode corretamente no ambiente do Heroku, precisamos instalar alguns módulos. Veremos cada um deles a seguir:

Gunicorn

Gunicorn é um servidor web de código aberto para Python. Ele permite que o Heroku implante nosso aplicativo em vários `Workers`, ou seja, ele vai rodar com mais de um processo simultâneo. Se tivermos três workers, nossa aplicação rodará em três instâncias simultâneas, permitindo que o sistema tenha uma performance melhor e consiga atender a uma quantidade maior de usuários.

Para instalá-lo, execute o seguinte comando em seu projeto Django:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install gunicorn
Collecting gunicorn
  Using cached gunicorn-20.0.4-py2.py3-none-any.whl (77 kB)
Requirement already satisfied: setuptools>=3.0 in
c:\users\tiago.ribeiro\documents\tiago ribeiro\livro_django\venv\lib\site-
packages (from gunicorn) (46.1.3)
Installing collected packages: gunicorn
Successfully installed gunicorn-20.0.4
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Procfile

Um Procfile é algo exclusivo do Heroku. É um arquivo no diretório raiz do seu projeto que informa ao Heroku como o aplicativo deve ser iniciado e executado, ou seja, o que deve acontecer com a

aplicação no momento em que você realiza o deploy. No nosso caso, usaremos nosso Procfile para solicitar que o Heroku execute um servidor Gunicorn e depois aponte esse servidor para o nosso projeto Django.

Crie um arquivo chamado `Procfile` dentro da raiz do projeto Django. Seu projeto ficará assim:

```
livro_django
|——Procfile
├——medicSearch
|        migrations
|        models
|        static
|        templates
|        tests
|        urls
|        views
|        __init__.py
|        ... Arquivos ocultos para otimizar a página
├——medicSearchAdmin
|        ... Arquivos ocultos para otimizar a página
```

Adicione o seguinte código ao Procfile:

```
web: gunicorn medicSearchAdmin.wsgi
```

Agora estamos dizendo para o Heroku iniciar nossa aplicação utilizando o módulo do Gunicorn. Isso fará com que o aplicativo Django rode dentro do Heroku.

Django-heroku

O Heroku criou um módulo chamado `Django-heroku` para facilitar um pouco nosso trabalho na hora de realizar o deploy. Quando digo facilitar o trabalho, refiro-me a uma série de configurações complexas que deveriam ser feitas manualmente e que, com essa biblioteca, não precisaremos fazer, pois ela já possui todas as *settings* necessárias para o sistema rodar pelo `Heroku` .

Vamos instalá-lo e configurá-lo conforme veremos a seguir. Rode o comando para instalar:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip install django-heroku
Collecting django-heroku
  Using cached django_heroku-0.3.1-py2.py3-none-any.whl (6.2 kB)
Requirement already satisfied: whitenoise in
c:\users\tiago.ribeiro\documents\tiago ribeiro\livro_django\venv\lib\site-
packages (from django-heroku) (5.1.0)
Requirement already satisfied: psycopg2 in
c:\users\tiago.ribeiro\documents\tiago ribeiro\livro_django\venv\lib\site-
packages (from django-heroku) (2.8.5)
Successfully installed django-heroku-0.3.1
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Agora que instalamos o módulo, vamos abrir o arquivo `settings.py` dentro da pasta `medicSearchAdmin/settings` e adicionar o código a seguir:

medicSearchAdmin/settings/settings.py

```
# Adicione no final do código

import django_heroku
django_heroku.settings(locals())
```

Em seguida, precisamos criar um arquivo chamado `production.py` dentro da pasta `medicSearchAdmin/settings/` . Faça isso e adicione o código a seguir:

medicSearchAdmin/settings/prodution.py

```
from .settings import *

DEBUG = True

# Crie a secret key para seu ambiente de produção
SECRET_KEY = 'ixb62hb#ts=ab532u%p1_62-!5w2j==j6d^2-j$!z(@*m+-h'

ALLOWED_HOSTS = ['*']
```

```
# Database
# https://docs.djangoproject.com/en/2.2/ref/settings/#databases
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Agora que o arquivo de `settings` foi alterado e já criamos o arquivo de produção, o Heroku está pronto para rodar nossa aplicação, porém ele ainda não sabe quais módulos são necessários para rodar o projeto, dentre os que vamos instalar na nuvem.

Vamos utilizar o comando `pip freeze > requirements.txt` para passar ao arquivo `requirements.txt`, que fica no diretório raiz, todas as libs que instalamos durante o desenvolvimento do projeto. Rode o comando:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ pip freeze > requirements.txt
```

Abra o arquivo `requirements.txt` e veja as bibliotecas nele:

```
asgiref==3.2.3
backcall==0.1.0
certifi==2020.4.5.1
cffi==1.14.0
chardet==3.0.4
colorama==0.4.3
coverage==5.2
... Linhas ocultas
```

Você verá algo parecido com o resultado anterior, porém com mais linhas que foram ocultadas aqui para não ocupar espaço.

> As seguintes bibliotecas podem acabar indo para o `requirements.txt` e gerar problemas no deploy: `ipython`, `ipython-genutils` e `traitlets`. Se elas estiverem no arquivo, remova-as.

Agora crie um arquivo na raiz chamado `.gitignore` e escreva o código a seguir:

```
### Django ###
*.log
*.pot
*.pyc
__pycache__/
```

O `.gitignore` vai evitar que arquivos com extensões .log, .pot, .pyc e pastas com nome `__pycache__` subam para o diretório do Heroku na nuvem. Pronto, estamos preparados para rodar nosso primeiro deploy. Vamos lá.

15.5 Login e deploy no Heroku

Vamos começar rodando o comando `heroku login` e realizando o login no CLI:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
```

Uma tela de login similar a esta imagem abrirá:

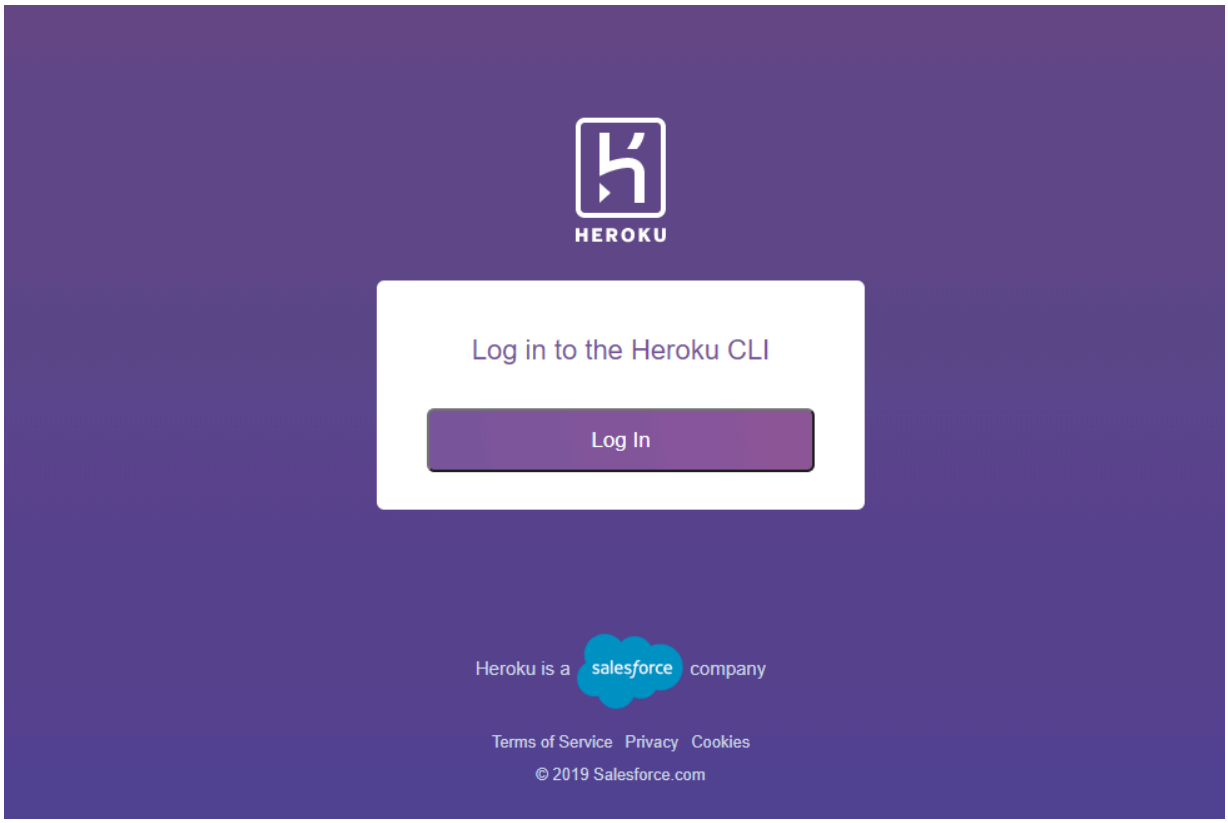


Figura 15.2: Login Heroku

Em alguns casos, se você já estiver logado, ao clicar em `Login In`, ele mandará você fechar a janela e voltar para o projeto. Caso não esteja logado em seu navegador, ele pedirá usuário e senha, e depois aparecerá a mensagem de fechar a janela do navegador. Será uma imagem assim:

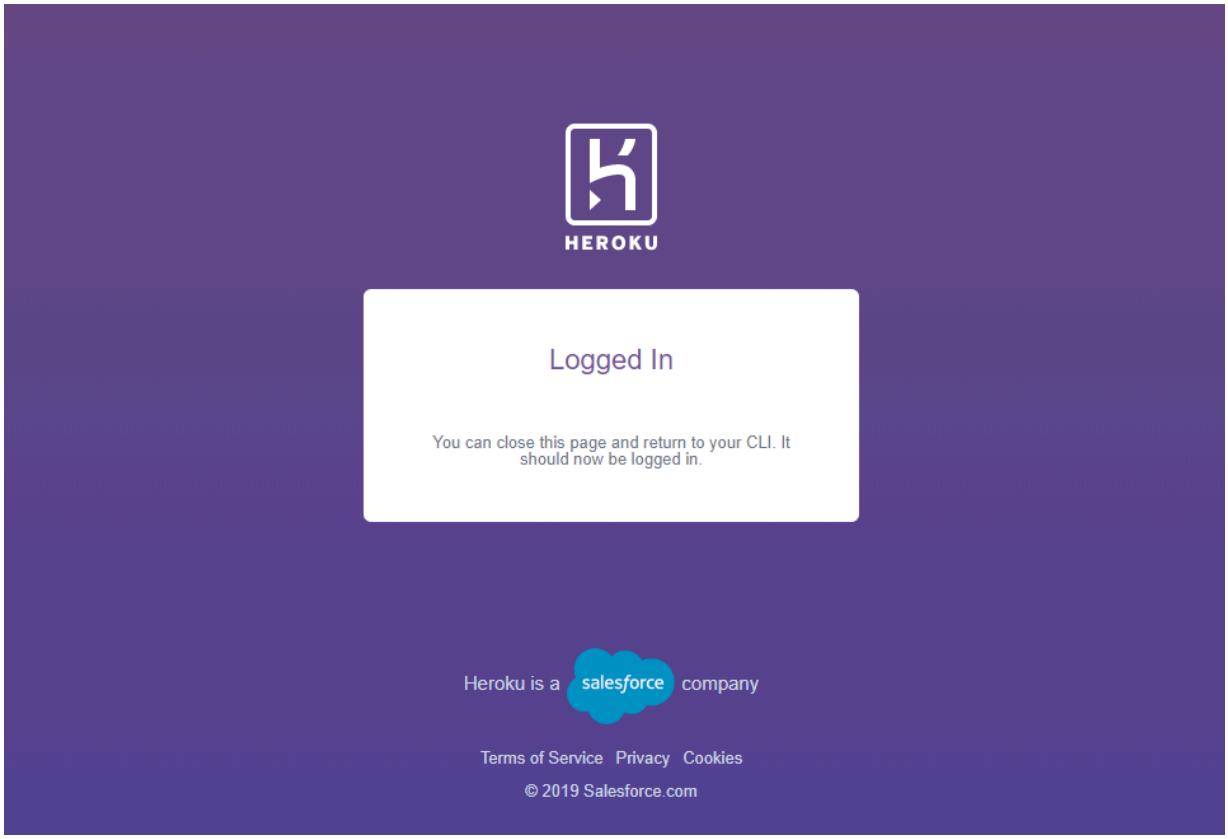


Figura 15.3: Login finalizado

Ao voltar para o projeto, o terminal ficará assim:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/50c241e2-c4c7-4238-8a83-0058ef0a0489
Logging in... done
Logged in as email@email.com (Aqui aparecerá seu e-mail)
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Agora vamos rodar o comando para criar um projeto no Heroku onde subiremos nosso app Django. Rode o comando `heroku create livro-django`, onde `livro-django` é o nome que você quer dar para o aplicativo. No caso, eu já estou usando `livro-django`, então você precisará colocar outro nome à sua escolha. Separe apenas por traço, não use underline, pois o Heroku não aceita:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ heroku create livro-django
Creating ⬢ livro-django... done
https://livro-django.herokuapp.com/ | https://git.heroku.com/livro-
django.git
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Com o projeto criado, vamos abrir o Heroku na web para fazer uma configuração antes de subir nossa aplicação:

Por padrão, o Heroku procura o arquivo `settings.py`. Como ele está incompleto, já que temos os arquivos `development.py`, `testing.py` e `production.py`, ocorrerá um erro, mas isso será facilmente corrigido. Para isso, vamos abrir nosso dashboard no navegador e adicionar essa variável ao ambiente.

Abra a url: https://id.heroku.com/login e faça login com suas credenciais.

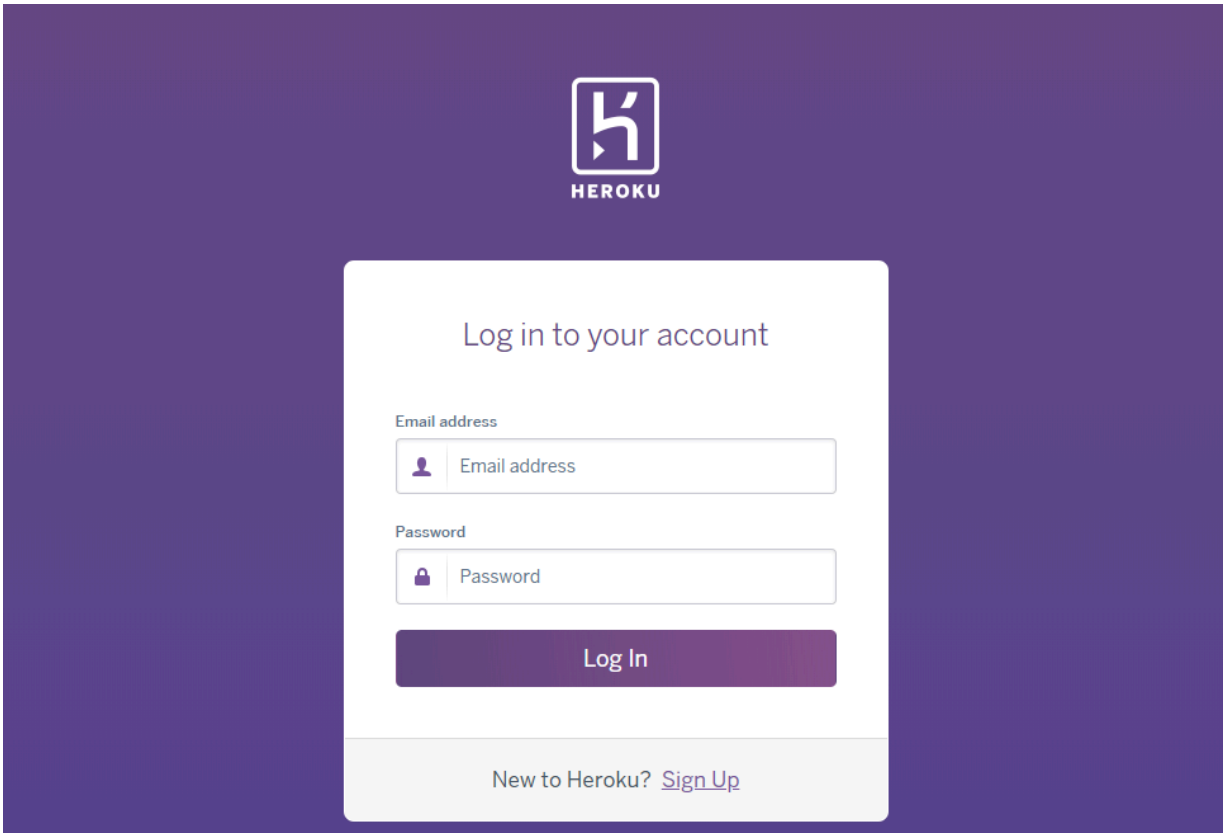


Figura 15.4: Login do Heroku no navagador

Clique no nome do projeto:

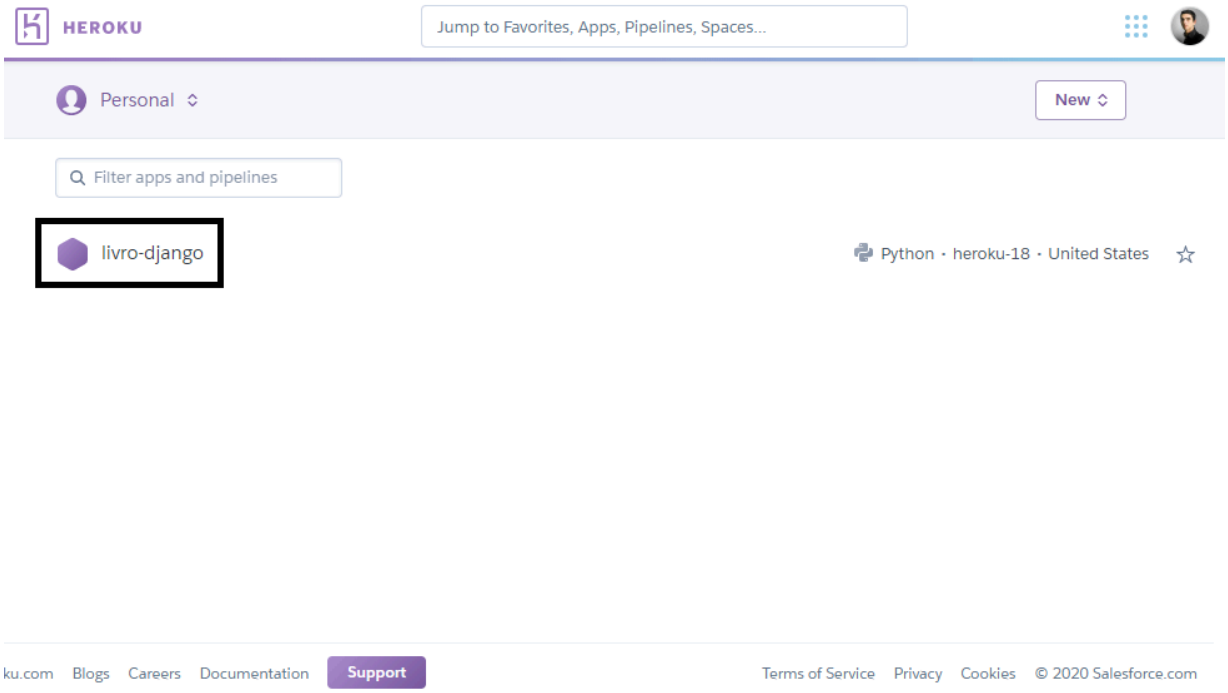


Figura 15.5: Dashboard do Heroku

Clique na opção `settings`:

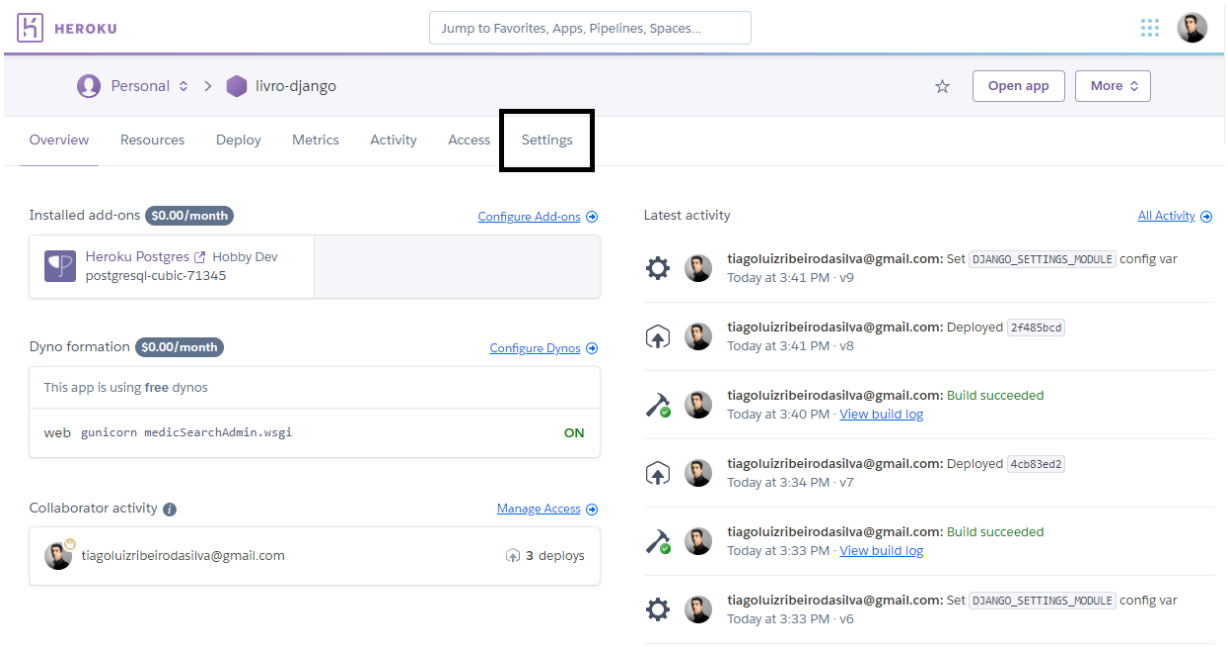


Figura 15.6: Dashboard do projeto

Clique no botão `Reveal Config Vars` :

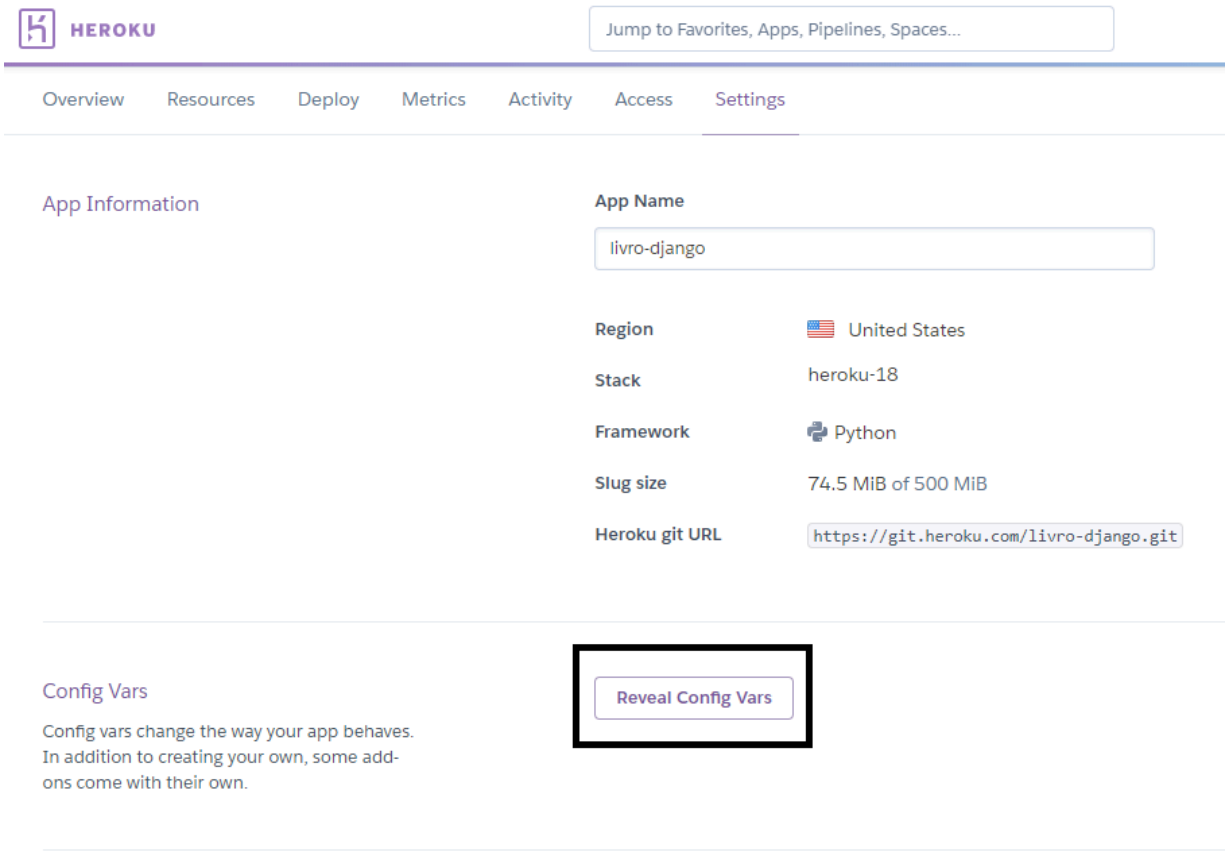


Figura 15.7: Exibir variáveis

Preencha os dois campos em branco com os valores:

- **key**: `DJANGO_SETTINGS_MODULE`
- **value**: `medicSearchAdmin.settings.production`

E depois clique em `Add` :

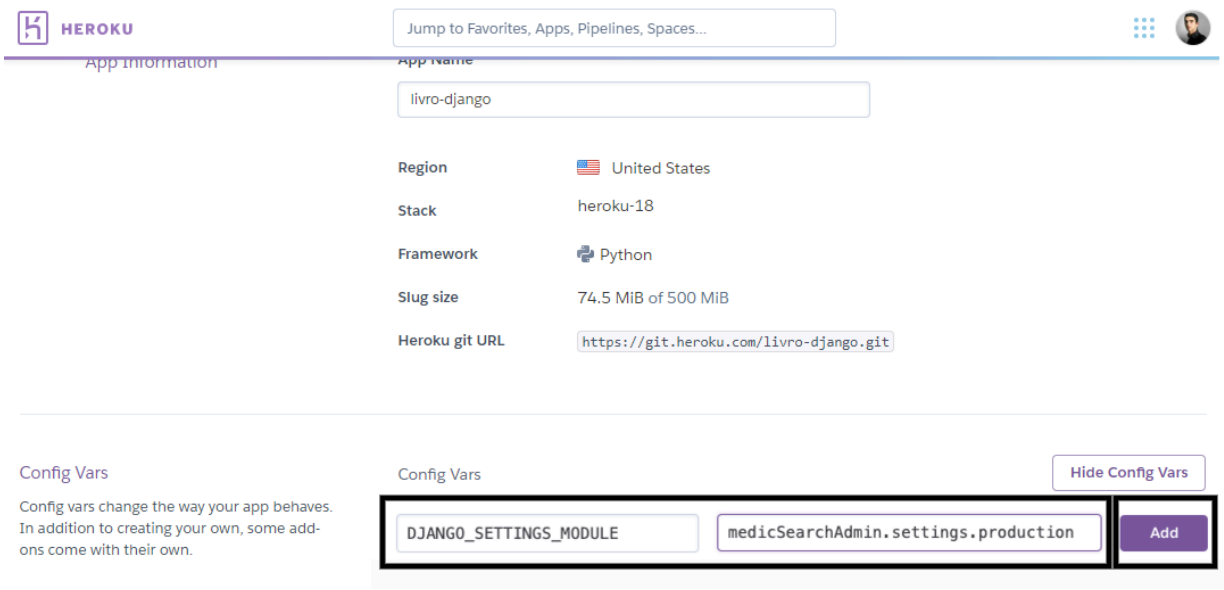


Figura 15.8: Adicionar variáveis

Isso fará com que o Heroku adicione a variável `DJANGO_SETTINGS_MODULE` com o valor `medicSearchAdmin.settings.production` à nossa máquina da nuvem, assim a aplicação terá esse arquivo como arquivo de configurações.

Agora que configuramos o que faltava, vamos realizar o deploy rodando a seguinte sequência de comandos:

- **git init**: comando que usamos para iniciar um repositório do GIT dentro do projeto.
- **heroku git:remote -a livro-django**: comando para dizer que esse repositório é o projeto `livro-django` que acabamos de criar. Fique atento ao nome que você deu ao seu projeto, ele deve substituir `livro-django` .
- **git add .** (Fique atento ao ponto, ele faz parte do comando add): comando para adicionar todos os arquivos do projeto para realizarmos o deploy.
- **git commit -am "Primeiro deploy da aplicacao"**: comando para gerar o pacote de commit que subirá para o Heroku.
- **git push heroku master"**: por fim, o comando que vai enviar nossa aplicação para o Heroku.

Quando o Heroku reconhecer que recebeu um commit novo da aplicação, ele vai instalar os módulos do arquivo `requirements.txt` e executar o comando `Procfile`, e com isso a aplicação vai rodar. Se tudo der certo você verá uma saída similar a esta aqui no terminal:

```
(venv) tiago_luiz@ubuntu:~/livro_django$ git push heroku master
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 74.5M
remote: -----> Launching...
remote:        Released v7
remote:        https://livro-django.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/livro-django.git
   f0d2455..4cb83ed  master -> master
(venv) tiago_luiz@ubuntu:~/livro_django$
```

Minha aplicação por exemplo roda em https://livro-django.herokuapp.com/

Figura 15.9: Aplicação do livro

Agora que você já sabe o endereço da sua aplicação, para deixá-la mais segura, abra o arquivo `production.py` dentro da pasta `medicSearchAdmin/settings/`. Faça uma pequena alteração removendo o valor `*` de dentro da lista de `ALLOWED_HOSTS` e coloque a url em que seu aplicativo está rodando no `Heroku`; no meu caso o valor será `livro-django.herokuapp.com`:

medicSearchAdmin/settings/prodution.py

```
from .settings import *

DEBUG = True
# Como estamos no Heroku o DEBUG precisará ficar True para a aplicação
rodar, em um ambiente de produção real desabilite o DEBUG.
```

```
SECRET_KEY = 'ixb62hb#ts=ab532u%p1_62-!5w2j==j6d^2-j$!z(@*m+-h'

# Altere a linha a seguir, colocando o endereço do seu app do Heroku.
ALLOWED_HOSTS = ['livro-django.herokuapp.com']

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Você pode deixar `*`, mas isso não é seguro, é melhor permitir que apenas o Heroku rode sua aplicação em produção. Isso evitará tentativas de fraude que possam ocorrer na sua plataforma. O Django é um dos frameworks mais seguros que existe exatamente por ter esse tipo de recurso, que permite sermos mais seletos nas configurações da nossa aplicação.

Com isso, concluímos o desenvolvimento do nosso sistema em Django, desde a criação da aplicação, passando pela criação de testes de unidade e seu deploy.

> Lembre-se de que o recomendado é criar sempre os testes antes das funcionalidades. Aqui foi necessário colocar o teste no final, pois precisávamos antes aprender a mexer com o Django, mas isso é recomendado apenas em casos didáticos como este.

Todas as vezes em que você precisar atualizar sua aplicação no Heroku, é só rodar os comandos:

- `git add`
- `git commit -am "Texto do commit (É só para identificar cada mudança)"`
- `git push heroku master`

Com isso você conseguirá manter seu app sempre atualizado. Se quiser olhar o log do app, caso ele dê erro, você poderá rodar o comando `heroku logs --tail` ou acessar pelo `dashboard`:

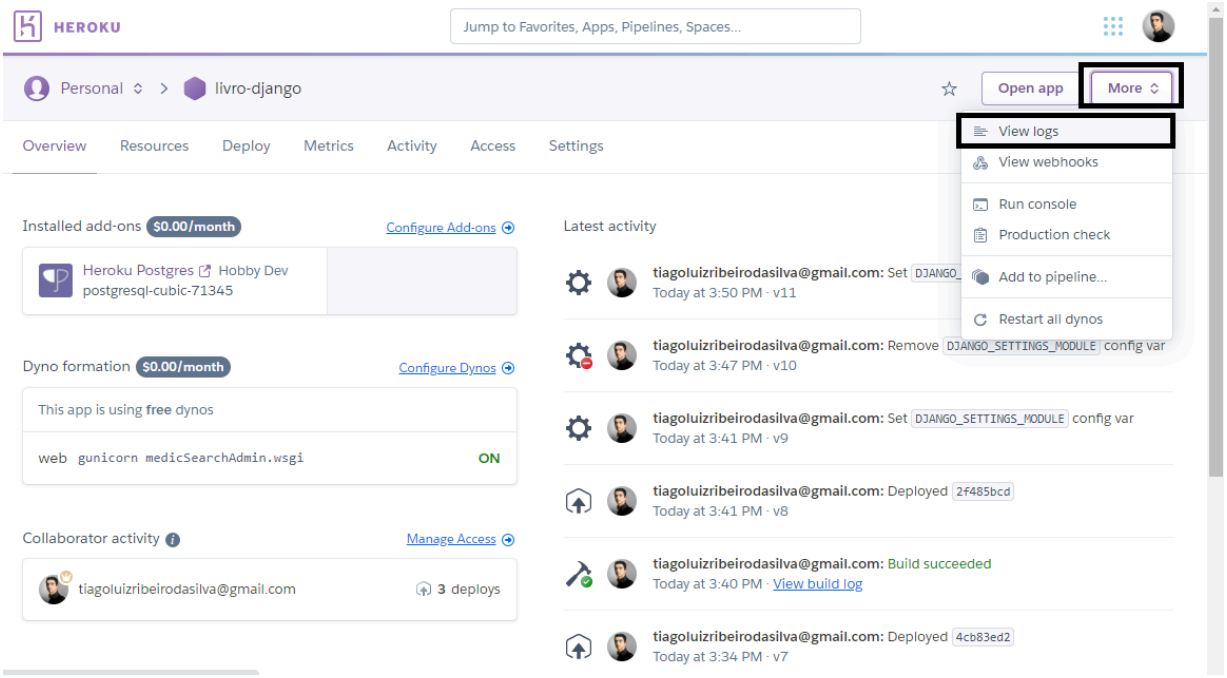


Figura 15.10: Logs do Heroku

O log aparecerá igual ao que é exibido em nosso terminal quando o projeto está rodando.

15.6 Situações específicas

Existe a possibilidade de o Heroku adicionar um banco de dados dele para usarmos e, caso ele faça isso automaticamente, teremos problemas na aplicação. Para remover esse banco de dados siga os passos a seguir:

Vá ao overview do projeto e clique em `Configure Add-ons`:

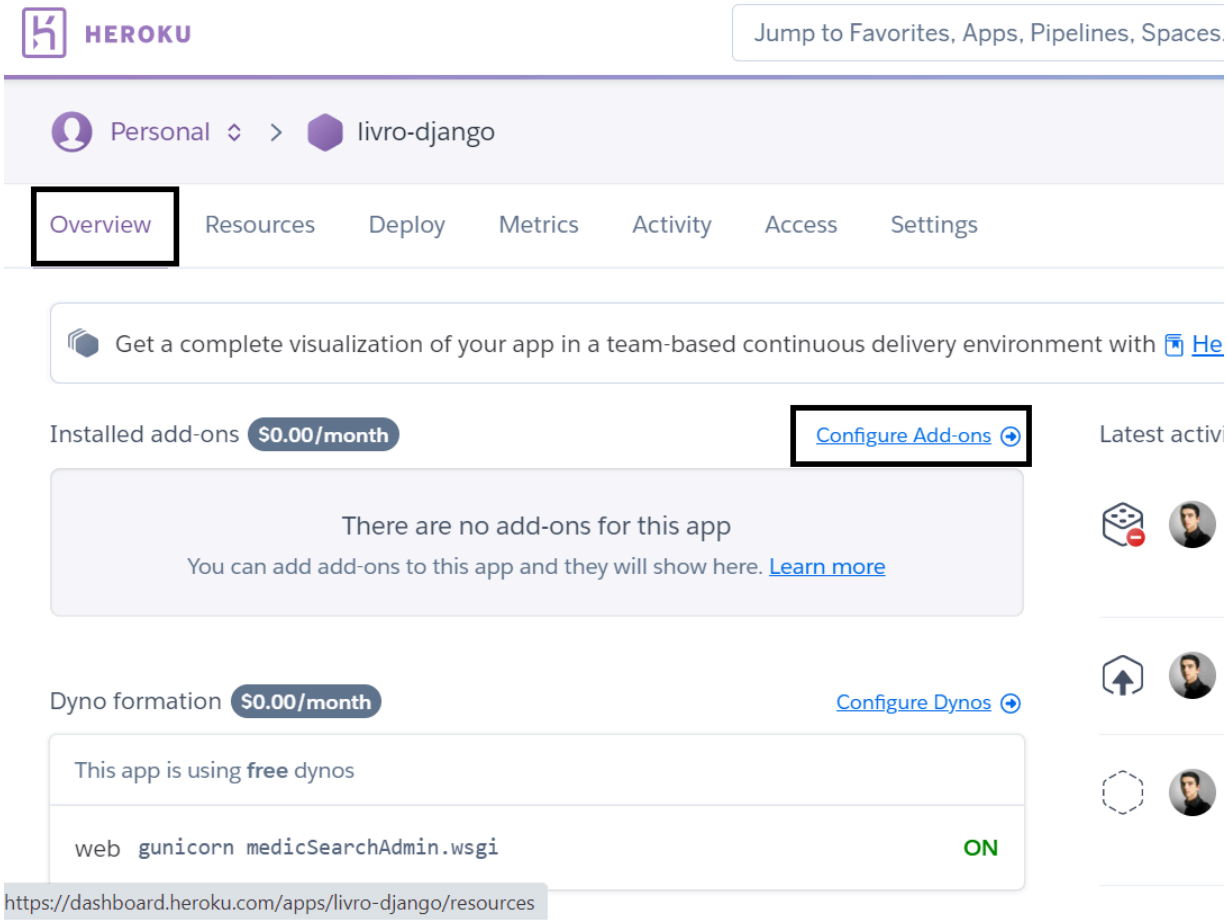


Figura 15.11: Passo 1 - banco

Caso haja um banco criado, selecione na área lateral direita do banco o menu de opções dele:

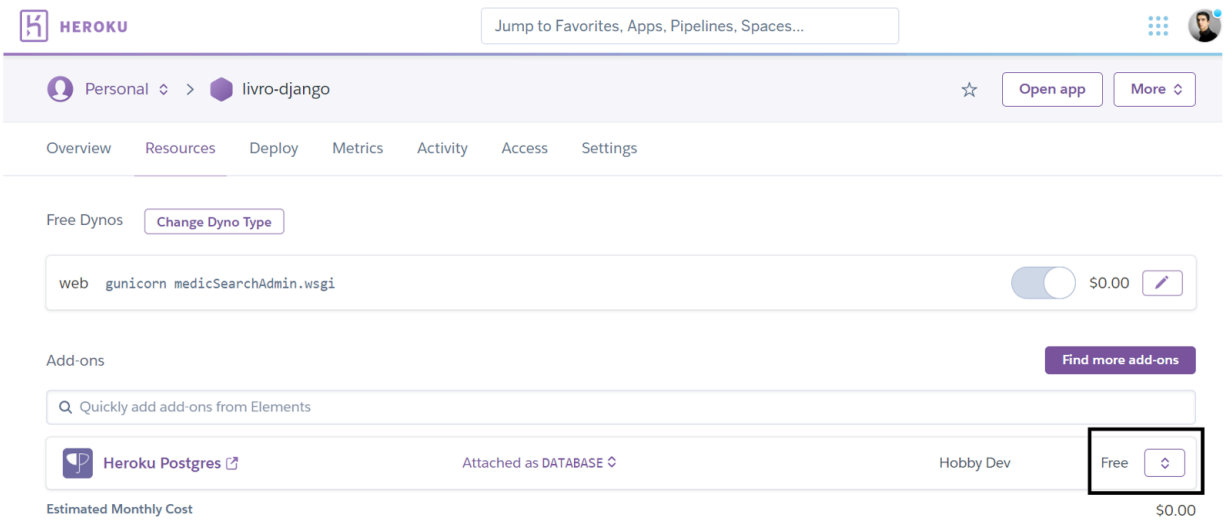


Figura 15.12: Passo 2 - banco

Clique na opção `Delete Add-on` :

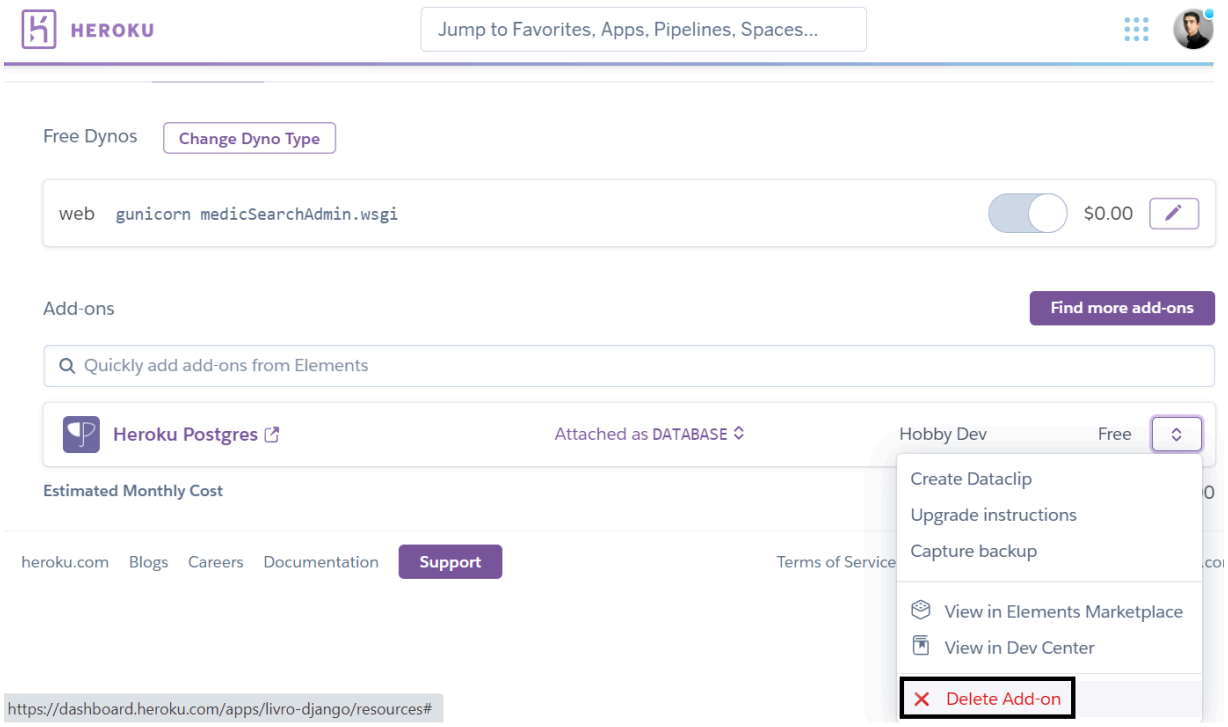


Figura 15.13: Passo 3 - banco

Adicione o nome do projeto para confirmar a remoção:

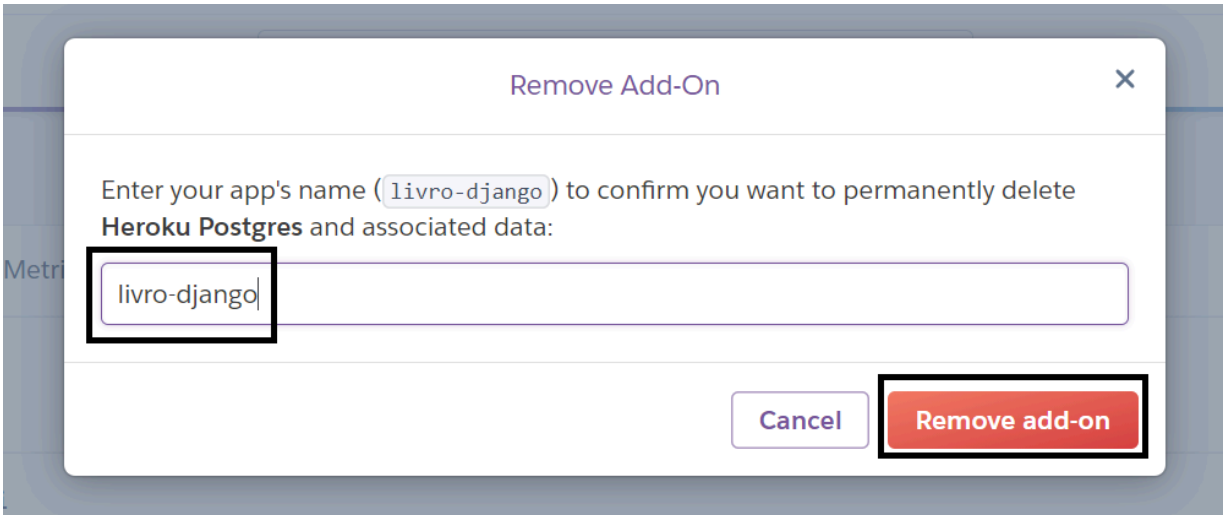


Figura 15.14: Passo 4 - banco

Pronto, problema resolvido.

Para acessar o código do projeto acesse a url https://github.com/tiagoluizrs/livro_django e clone a aplicação para comparar com o projeto que você está desenvolvendo. Espero que este livro tenha ajudado você a alcançar seus objetivos com o Django.

Fique com Deus e até a próxima!!!